ALGOL 68

EINFÜHRUNG IN DIE ALGORITHMISCHE
SPRACHE ALGOL68

BCCU

1. GRUNDBEGRIFFE UND BEISPIELE

1.1 Basis und Erweiterungsprinzip

1.1.1 Ausgangsbeispiel
1.1.2 Die Grundmengen
1.1.3 Der Begriff "orthogonal design"

1.2 Modell einer ALGOL68-Maschine

1.2.1 Externe und interne Objekte
1.2.2 Relationen zwischen externen Objekten
1.2.3 Relationen zwischen internen Objekten
1.2.4 Der Begriff Variable (die Relation "to refer to")
1.2.5 Relationen zwischen einem externen und einem internen Objekt
1.2.6 Lebensdauer von internen Objekte (der Begriff "scope")

1.3 Abarbeitung eines ALGOL68-Programms

1.3.1 Die assignation (Zuweisung)
1.3.2 Vergleich von Werten
1.3.3 Routinen und Operatoren
1.3.4 Standard-Prozeduren und Standard-Operatoren
1.3.5 Serielle und kollaterale Abarbeitung

1.4 Programm-Beispiele, verschiedene Schreibweisen

1.4.1 Komprimierte Schreibweise
1.4.2 Label, *goto* und conditional-clause
1.4.3 Loop-clauses
1.4.4 Reihen ("multiple values")
1.4.5 Strukturen ("structured values")
1.4.6 Prozeduren ("routines")
1.4.7 Verschiedene Darstellungsformen
1.4.8 Die particular-prelude dieses Buches

---

1. GRUNDBEGRIFFE UND BEISPIELE

1.1. Basis und Erweiterungsprinzip

1.1.1. Ausgangsbeispiel

Als erstes Beispiel betrachten wir folgendes Programm, das so gewählt wurde, dasz es auch ein ALGOL60-Programm darstellt.
Mit Hilfe dieses Programms werden zwei Zahlen gelesen und sodann ihr Mittelwert sowie ihre Standard-Abweichung ausgedruckt. Selbstverständlich wird dies erst dann sinnvoll, wenn mehrere Zahlen gelesen und behandelt werden (vgl. dazu {2} und {3}).

```
{1} begin   co Mittelwert und Standard-Abweichung
                zweier Zahlen co

            real x,y, sum, squm,sd;
            read(x); read(y);
            sum := x+y ;
            squm:= x↑2+y↑2 ;
            sd  := sqrt(2×squm-sum↑2)/2 ;
            print(sum/2); print(sd)

    end
```

Hierbei haben die identifiers *x*, *y*, *sum*, *squm* und *sd* ihre Bedeutung durch die declaration

```
real x,y, sum,squm,sd
```

erhalten.
An diesem Beispiel (und auch an den interessanteren Beispielen {2} - {7}, vgl. 1.4) werden wir in den folgenden Paragraphen einige Grundbegriffe und Prinzipien der Sprache ALGOL68 kennenlernen.

1.1.2. Die Grundmengen

ALGOL68 kann man mit einem LEGO-Spiel vergleichen: Als Basis sind Bausteine verschiedener Form und gewisse Anbaumöglichkeiten fest vorgegeben. Hierbei gibt es nur wenig voneinander verschiedene Formen von Bausteinen, und ihre Anbaumöglichkeiten sind unabhängig von der Form der Bausteine.
Die fest vorgegebenen Bausteine sind Werte ("values") von nur wenigen verschiedenen Typen. Diese Werte werden also aufgeteilt in eine kleine Anzahl von Grundmengen. Die Grundmengen sind:

boolean , character ,

integral , real ,

format .

Wir werden später sehen, dasz man die leere Menge (void) auch noch als eine 'Grundmenge' betrachten kann: eine Menge also, die keine Werte enthält.
Die einzelnen Elemente der (nicht-leeren) Grundmengen (also: die Grundwerte) werden mit denotations bezeichnet, wie z.B.

true , *"a"* ,

37 , *3.1415926536* ,

$+d.13de1d$

und die Mengen selbst mit mode-indications:

void ,

bool , *char* ,

int , *real* ,

format .

Die Grundmengen, ihre indications und die Bezeichnung (denotation) ihrer Werte sind in der nachfolgenden Tabelle angegeben.

indications	"values"	denotations
void	leer	keine
bool	die Menge, bestehend aus den beiden logischen Werten"wahr" und "falsch"	*true* *false*
char	eine Mindestmenge von Zeichen: 1 Alphabet von Buchstaben, die Ziffern *0* bis *9*, einige Sonderzeichen, abhängig von der jeweiligen Implementierung sind , eventuell noch andere Zeichen (z.B. noch ein Alphabet) vorhanden	*"a" "b" "c" "d" ----* *------ "x" "y" "z"* *"0" "1" "2" ------ "9"* *". " ", " "+" "-" "="* *"(" ")" ":" ------* *"A" "B" "C" "D" ----* *------ "X" "Y" "Z"* *"/" "?" "!" "×" "&"* *"$_{10}$" ";" "%" ------*
int	ein Teilbereich der ganzen Zahlen (dessen Grenzen implementationsabhängig gegeben sind durch *max int*)	*0 1 2 3 4 5 6 7 8 9* *10 11 12 13 14 ----* *100 101 ----------* *--------- 67108863*
real	eine Teilmenge der reellen Zahlen, gegeben in Gleitpunkt-Annäherung; die kleinste positive reelle Zahl, die man noch mit Erfolg zu *1.0* addieren kann, ist gegeben durch *small real*, die absolut grösste reelle Zahl durch *max real*.	*3.14159265* *0.314159265$_{10}$1* *0.000314159265$_{10}$4* *31415.9265$_{10}$-4* *314159265$_{10}$-8* *0.0 1.0 0.1* *10$_{10}$10 1$_{10}$-100* *0.314159265e1* *314159265e-8*
format	eine Menge von 'strings' mit eigener Syntax für die Beschreibung von Ein- und Ausgabedaten.	*$+d.13de1d$* *$"dm"8d.2d$*

1.1.3. Der Begriff "orthogonal-design"

Mit Hilfe der wenigen (in 1.1.2. genannten) Grundbausteine kann man gröszere Bau-Elemente bilden, z.B. Reihen (eine gewisse Anzahl numerierter Elemente eines bestimmten Typs), Strukturen (ein Gebilde von Elementen, möglicherweise verschiedenen Typs), Routinen (Programmteile mit oder auch ohne gewisse Parameter, die dann nach ihrer Ausführung einen bestimmten Baustein liefern), Adressen (Werte die sich auf andere Bausteine beziehen), etc.
Aus diesen so zusammengesetzten Bausteinen können dann wiederum 'höhere' Bausteine gebildet werden (Reihen von Strukturen, Strukturen mit Reihen als Komponenten, Reihen von Routinen, Routinen, die jeweils eine Reihe oder ein Struktur liefern, etc.).
Dieses Erweiterungsprinzip kann man beliebig oft und in jeder Kombination anwenden. Die Generalisierung des ALGOL60-Begriffs Typ heiszt in ALGOL68: "mode". Einen gewünschten neuen "mode" kann man aus den Grund-"modes" (*bool*, *char*, *int*, *real* und *format*) bilden mit Hilfe von gewissen symbols (die wir oft als 'mode-maker' bezeichnen werden).
Die 'mode-maker' sind:

[ ]	zur Konstruktion einer Reihe,
struct	zur Konstruktion einer Struktur,
flex	zur Konstruktion flexibler Grenzen in einer Reihe,
long	zur Konstruktion gröszerer Genauigkeit (von *int*, *real*, etc.),
ref	zur Konstruktion einer Adresse,
proc	zur Konstruktion einer Routine,
union	zur Konstruktion einer Vereinigung mehrerer "modes"

Die "modes" (also: die Mengen) werden dabei durch sogenannte declarers spezifiziert. So spezifiziert z.B:

[*1:n*]*real*

den "mode" row-of-real, d.i. der "mode" einer Reihe von *reals*.
Einen neuen indication (für die Bezeichnung einer Menge) kann man durch eine mode-declaration definieren: rechts von dem is-defined-as-symbol "=" steht der declarer.

Beispiele:

```
mode boolean = bool
mode logical = bool
```

definieren *boolean* oder *logical* als alternative indications für *bool*.

```
mode vector  = [1:n]real
mode matrix  = [1:n,1:n]real
```

definieren *vector* und *matrix* als ein- bzw. zweidimensionale Reihe von *real*s.

```
mode compl   = struct(real re,im)
```

definiert *compl* als eine Struktur mit zwei *real*-Komponenten, die man mit den field-selectors *re* und *im* auswählt.

```
mode string  = flex[1:0]char
```

definiert *string* als eine *char*-Reihe mit flexiblen Grenzen.

```
mode giant   = long int
```

definiert *giant* als einen *int*, wobei *max long int* gröszer (oder, implementationsabhängig, wenigstens gleich) *max int* ist.

```
mode man     = struct(string name, int year,
                      ref woman wife) ,
mode woman   = struct(string name, int year,
                      ref man   husband,
                      ref flex[1:0]human child) ,
mode human   = union(man,woman)
```

definieren ein naives Modell der Menschheit, wobei die elementare Tatsache, dasz eine Frau 0,1,2,--- Kinder gebären kann, ein Mann jedoch nicht, zum Ausdruck gebracht worden ist und dasz ein *human* entweder ein *man* oder eine *woman* sein kann.

```
mode fun     = proc(real)real
```

definiert *fun* als eine reelle Funktion auf der Menge der reellen Zahlen.

```
mode matmat  = proc([,]real)[,]real
```

definiert *matmat* als eine Funktion die Matrizen in Matrizen überführt; oder auch:

```
mode matmat  = proc(matrix)matrix
```

Solche mode-declarations sind nur dann notwendig, wenn "modes" miteinander verknüpft sind, wie es bei *man*, *woman* und *human* der Fall ist. Somit kann man ein indication als eine Abkürzung für einen zusammengesetzten declarer ansehen. Allerdings, einen zusammengesetzten declarer kann man auch selbst im Programm benutzen.
Für die Werte in zusammengesetzten Mengen kann man ebenso zusammengesetzte denotations bilden:

declarer (bzw. indication)	denotation
(1:4)real	*(1.0 , 5.1 , 0.2 , 6.3)*
(1:4,1:4)real	*((1.0 , 5.1 , 0.2 , 6.3) ,* *(0.0 , 1.0 , 4.2 , 1.9) ,* *(0.0 , 0.0 , 1.0 , 8.3) ,* *(0.0 , 0.0 , 0.0 , 1.0))*
compl	*(1.0 , 2.1)*
string	*"this is a row-of-character"*
giant	*long 2846 3829 8567 7200* *long 0*
woman	*("juliet capulet" , 1473 , romeo ,* *nil)*
fun	*(real x) real: sin(x)/x*

Sowohl mit den Grundbausteinen als auch mit den aus ihnen zusammengesetzten Bausteinen kann man Aktionen ("actions") definieren. Auch hier gibt es nur wenige Grundaktionen, wie: zuweisen (assignation), vergleichen, subtrahieren, indizieren, selektieren, *int* zu *real* und *real* zu *compl* erweitern ("widening") und noch einige mehr.
Mit Hilfe dieser Grundaktionen kann man dann zusammengesetzte Aktionen definieren, ähnlich wie man aus den Grundbausteinen zusammengesetzte Bausteine bildet. Aus diesen zusammengesetzten Aktionen kann man dann weiter "höhere" Aktionen bilden, usw.
Das Ganze ist in ALGOL68 so organisiert, dasz die Erweiterungsmöglichkeiten fast keine Ausnahmen kennen und dasz die Grundbegriffe (Elementar-Bausteine und Elementar-Aktionen) in ihrer Zahl

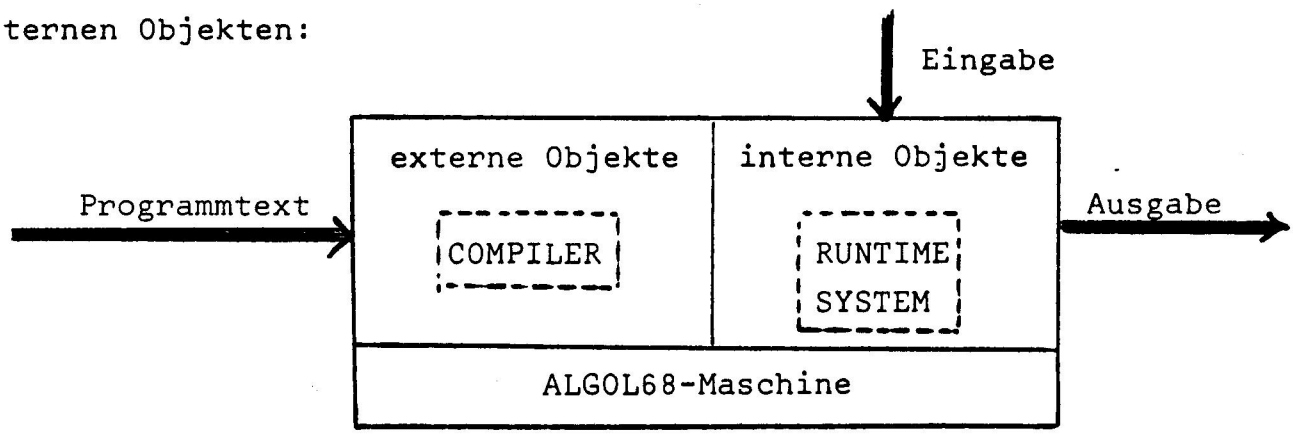
minimiert worden sind.
Dies zusammen bezeichnet man als "orthogonal design":

> das Leitmotiv in ALGOL68 und als solches
> viel wichtiger als die Sprache selbst.

1.2. Modell einer ALGOL68-Maschine

1.2.1. Externe und interne Objekte

Die Abarbeitung eines Programms wird im Report anhand eines idealisierten Computers beschrieben, den man die 'ALGOL68-Maschine' nennen kann.
In dieser Maschine wird nicht explizit zwischen 'compiletime' und 'runtime' unterschieden, wohl aber zwischen externen und internen Objekten:



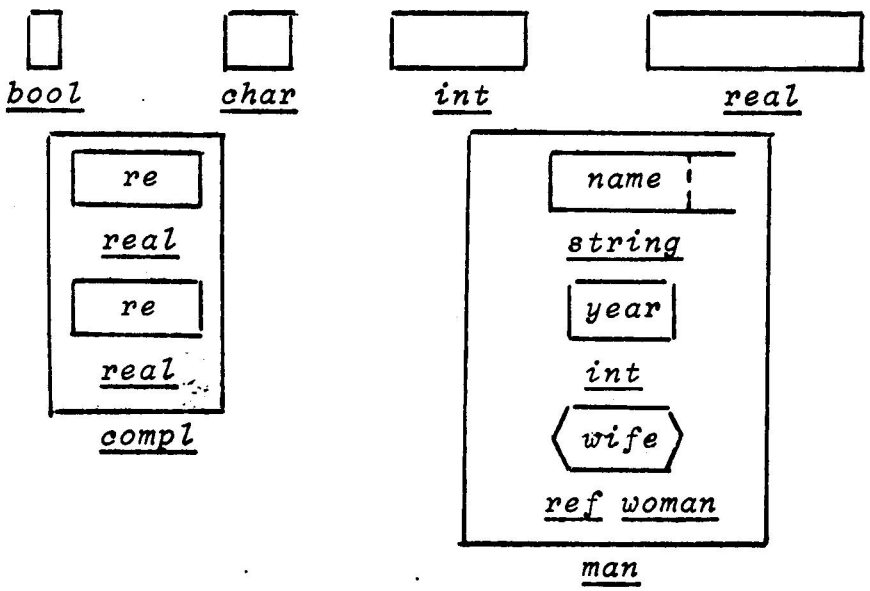
Die Abarbeitung eines externen Objekts, speziell eines Programms, wird im Report als "elaboration" bezeichnet. Auch wir werden diesen Begriff oft benutzen.
Ein externes Objekt ("construct") ist ein von der Syntax bestimmter Teil des Programmtextes.
Die interne Objekte sind die Werte im Maschinenspeicher. Für solch ein Wert gilt immer zweierlei:

a) er gehört zu einer bestimmten Menge: er hat einen bestimmten "mode",

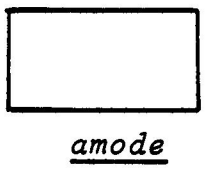
b) er tritt an einer bestimmten Stelle ("locale") im Speicher auf, hat demzufolge eine Adresse.

Die "elaboration" eines Programms durch die ALGOL68-Maschine geschieht durch Interpretation der in dem Programm enthaltenen externen Objekte, die zu gewissen Aktionen auf internen Objekten führt. Zwischen den (externen und internen) Objekten sind gewisse Relationen definiert, die bei der Interpretation eine wichtige Rolle spielen.

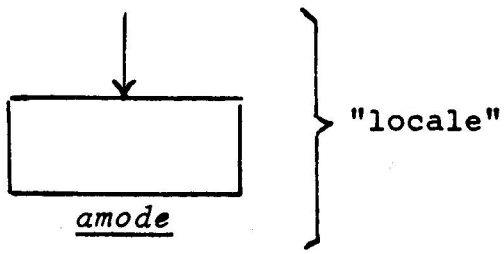
Wir werden die Wirkungsweise der ALGOL68-Maschine durch Bilder erläutern, die jeweils eine Momentaufnahme des Speicherinhaltes und der Relationen zwischen Objekten darstellen.
Dabei stellen wir einen Wert durch eine 'box' dar, deren Form seinem "mode" entspricht. Es repräsentieren dann 'boxes' verschiedener Form jeweils Werte, die verschiedenen Mengen angehören (also verschiedenen "mode" haben):



Für einen Wert irgendeines "mode" *amode* also:



Die Adresse, die ein Wert im Speicher hat, wird durch einen Pfeil wiedergegeben:



Das Paar, bestehend aus einer 'box' und seiner Adresse, entspricht ziemlich genau dem Begriff "locale", wie er im Report verwendet wird.

1.2.2. Relationen zwischen externen Objekten

Die beiden wichtigsten Relationen zwischen externen Objekten sind:

a) "enthalten" ("to contain")

b) "identifizieren" ("to identify")

Beispiele:

a1) identifier-declaration

real — declarer; *x* — identifier

(Für die besondere Schreibweise von externen Objekten vergleiche 2.1.2)

Hierbei sind die beiden externen Objekte declarer und identifier in dem externen Objekt identifier-declaration enthalten.

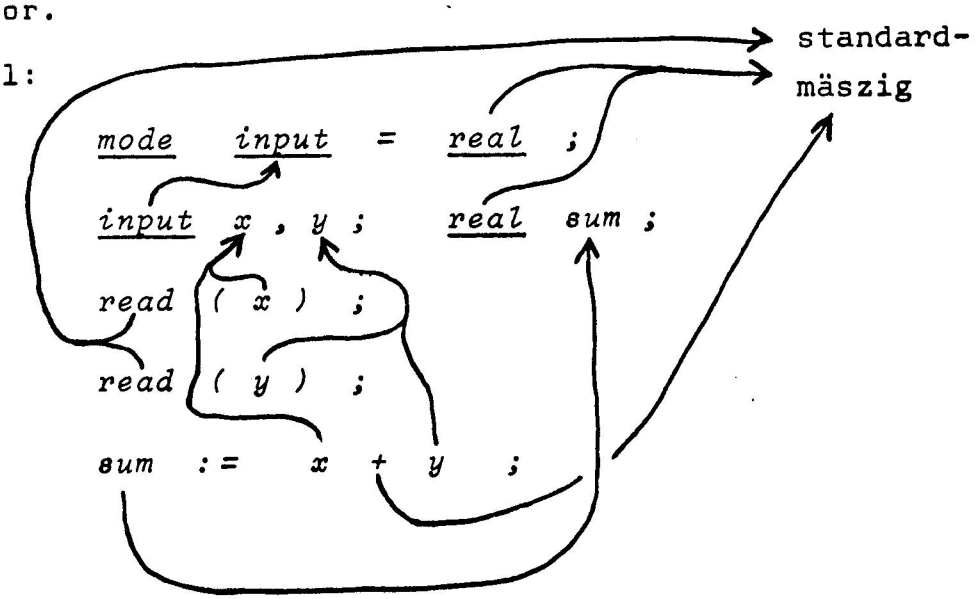
a2) assignation

sum — destination; *:=* — becomes-token; *x + y* — source

Hierbei sind die externen Objekte destination, becomes-token und source in dem externen Objekt assignation enthalen.

Sehr wichtig ist für uns die Relation "identifizieren". Diese Relation besteht zwischen speziellen externen Objekten, nämlich indicators. Wenn ein indicator in einer declaration enthalten ist, so bezeichnen wir ihn als defining-indicator. Tritt er dagegen sonst irgendwo im Programm auf, so wird er als applied-indicator bezeichnet.
Es identifiziert ein applied-indicator stets seinen defining-indicator.

Beispiel:



Hierbei wird die Relation "identifizieren" jeweils durch einen Pfeil angedeutet.

Wir werden später sehen (vgl. 2.4.2), dasz einem indicator durch eine declaration seine Bedeutung beigegeben wird. Durch die Identifizierung erhält ein indicator, wenn er als applied-indicator auftritt, stets die Bedeutung seines defining-indicator.

1.2.3. Relationen zwischen internen Objekten

Zwischen Werte (also internen Objekten) können folgende Relationen definiert sein:

a) "to be of the same mode as" (vom gleichen "mode" sein)
b) "to be equivalent to" (gleich sein)
c) "to be smaller than" (kleiner sein als)
d) "to refer to" (beziehen auf)

In diesem Paragraph betrachten wir die Relationen a)-c).
Sind diese Relationen zwischen zwei Werte definiert, so treffen sie für diese beiden Werte zu oder nicht.

a) Da alle Werte einen bestimmten "mode" haben (also jeweils zu einer bestimmten Menge gehören), so ist die Relation a) zwischen je zwei Werten definiert und man kann stets entscheiden, ob zwei Werte vom gleichen "mode" sind oder nicht ("conform" sind oder nicht).
So gilt zum Beispiel:

der *int*-Wert 1 ist vom gleichen "mode" wie der *int*-Wert 2
der *real*-Wert 1.0 ist vom gleichen "mode" wie der *real*-Wert 2.0
der *int*-Wert 1 ist nicht vom gleichen "mode" wie der *real*-Wert 1.0

b) Da Werte aus verschiedenen Mengen (also von verschiedenem "mode") gleich sein können, so kann zwischen ihnen die Relation "to be equivalent to" definiert sein. So ist diese Relation u.a. zwischen einem *int* und einem *real* definiert.
Zum Beispiel:

der *int*-Wert 1 ist gleich dem *real*-Wert 1.0
der *real*-Wert 0.0 ist gleich dem *int*-Wert 0

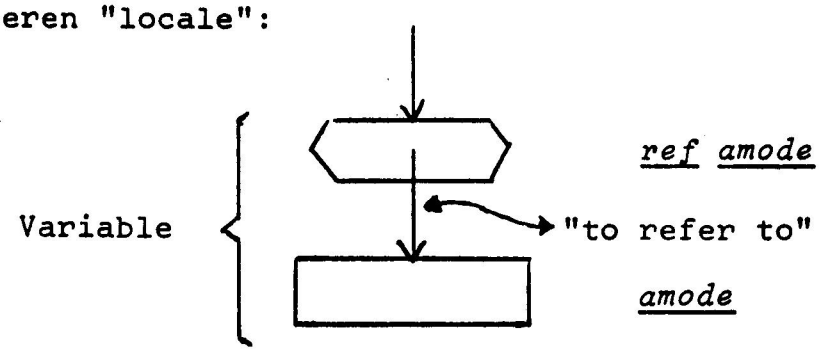
c) In bestimmten Mengen kann zwischen je zwei Elementen (also Werten von gleichem "mode") die Relation "to be smaller than" definiert sein. Diese Relation ist u.a. auf der Menge der *int*s und auf der Menge der *real*s definiert.
So gilt zum Beispiel:

der *int*-Wert 1 ist kleiner als der *int*-Wert 2
der *real*-Wert 3.1415926 ist kleiner als der *real*-Wert 3.1415927
der *real*-Wert 2.718 ist nicht kleiner als der *real*-Wert 2.718

1.2.4. Der Begriff Variable (die Relation "to refer to")

Wie in 1.2.1 angegeben, verstehen wir unter einem "locale" stets ein Paar, bestehend aus einer Adresse und einem Wert (dargestellt durch eine 'box').

In ALGOL68 kann eine Adresse selbst ein Wert sein. Sie kann dann selbst im Speicher auftreten und ist somit Bestandteil eines anderen "locale":



Wie aus diesem Bild schon deutlich wird, kann also ein Wert, wenn er eine Adresse ist, sich auf einen anderen Wert beziehen ("to refer to"). Der "mode" einer Adresse, die sich bezieht auf einen *amode*, ist *ref amode*.

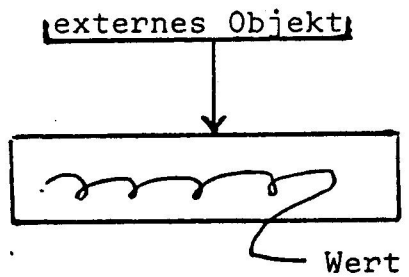
Nur dann, wenn man die Adresse eines *amode* als Wert zur Verfügung hat, kann man einen *amode* ändern.

> Deswegen nennt man das Paar, bestehend aus einer Adresse (*ref amode*) und dem Wert (*amode*), auf den sich diese Adresse bezieht, eine Variable.

1.2.5. Relationen zwischen einem externen und einem internen Objekt

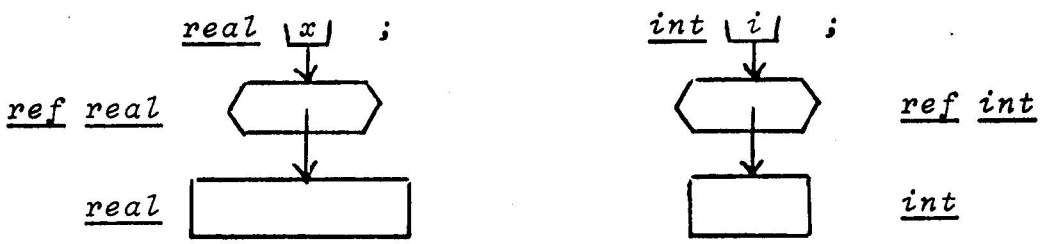
Durch die "elaboration" von externen Objekten können Werte im Speicher geliefert werden. Diese Werte werden im Report als die "yields" der "elaboration" bezeichnet.
Wird ein Wert speziell durch die "elaboration" einer declaration geliefert (vgl. 2.4.1), so erhält der in dieser declaration auftretende indicator diesen Wert, zu dem er dann Zugriff hat.
Diese Beziehung zwischen dem indicator, aufgefaszt als externes Objekt, und einem Wert, aufgefaszt als internes Objekt, heiszt "to access".

Wenn ein externes Objekt einen Wert erhält, so wollen wir dies im Bild folgendermaszen zum Ausdruck bringen:



Dabei kann ein und dasselbe externe Objekt verschiedene Werte erhalten, abhängig vom Kontext, in dem es auftritt.

Beispiele:

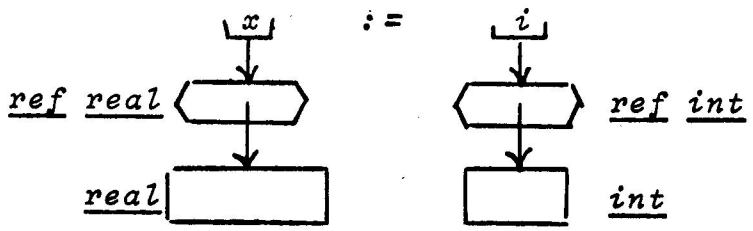


Hierbei erhalten die beiden identifier *x* und *i* als defining-identifier jeweils den Wert, den die "elaboration" ihrer declaration *real* *x* bzw. *int* *i* liefert, also einen *ref* *real* bzw. einen *ref* *int*. Die identifier *x* und *i* erhalten durch ihre declaration also beide eine Variable (*x* eine *real*-Variable und *y* eine *int*-Variable).

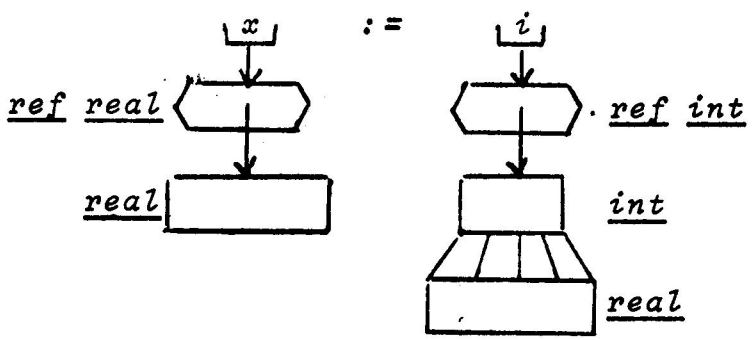
Betrachten wir sodann die assignation:

$$x := i$$

so identifizieren x und i als applied-identifier ihre defining-identifier (vgl. 1.2.2) und erhalten somit als Werte die ihrer defining-identifier:



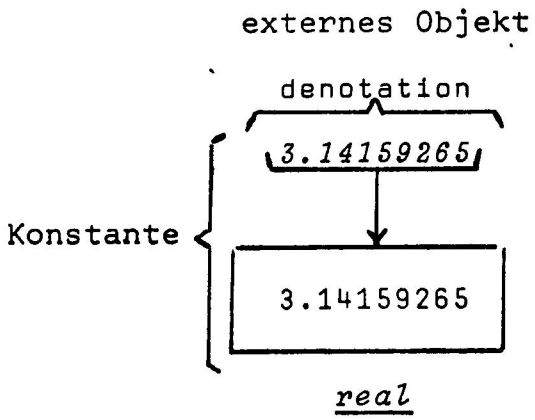
Im Kontext der assignation musz aber der applied-identifier i als rechte Seite dieser assignation ein *real* erhalten, das der Adresse von x (jetzt aufgefaszt als linke Seite) zugewiesen werden soll. Als defining-identifier hat i Zugriff zu einem *ref int*, der sich bezieht auf ("refers to") einen *int*; dieser wird dann 'erweitert' zu einem *real* (vgl. 4.2.4 für das "widening").
Somit erhalten wir folgendes Bild:



Die linke Seite (die destination) erhält also einen *ref real* (die Adresse eines *real*) und fordert somit einen *real*. Die rechte Seite erhält (liefert) nach einiger Bearbeitung einen *real*.

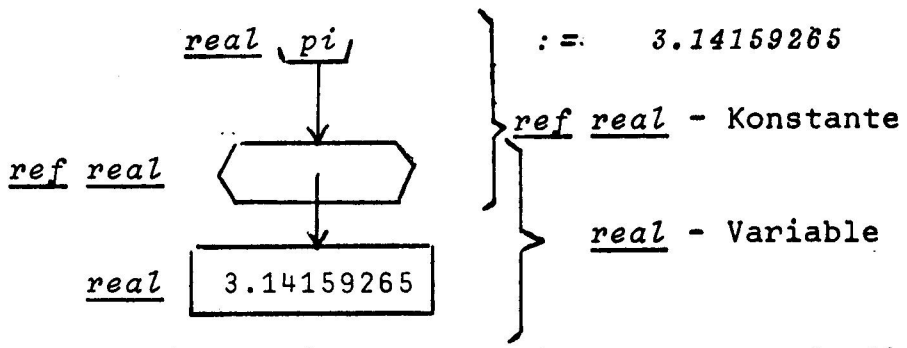
Wie wir in 1.1.4 gesehen haben, ist eine Variable ein Paar, bestehend aus einer Adresse (wenn sie selbst ein Wert ist) und dem Wert, auf den sich diese Adresse bezieht.
In Analogie dazu können wir das Paar, bestehend aus einem externen Objekt und dem Wert, zu dem dieses externe Objekt Zugriff ("access") hat, als eine Konstante bezeichnen.

Zum Beispiel:



Man beachte hier, wie *3.14159265* als externes Objekt Zugriff hat zu dem Wert 3.14159265 als internes Objekt (ein *real*) im Speicher.

Man vergleiche dieses Beispiel mit dem folgenden:



Wir sprechen hierbei von einer *ref real*-Konstanten, da die Adresse selbst nicht geändert werden kann.
Dagegen kann aber der *real*-Wert, auf den sich diese Adresse bezieht (z.B. durch eine Zuweisung) geändert werden, wir also von einer *real*-Variable sprechen können.

Eine Adresse, die selbst ein Wert im Speicher ist (wie im obigen Beispiel) heiszt im Report: ein "name".

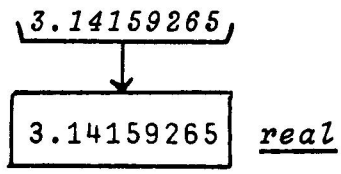
1.2.6. Lebensdauer von internen Objekten (der Begriff "scope")

Ein Wert (also ein internes Objekt) hat stets einen bestimmten "mode" und tritt als Bestandteil eines "locale" an einer bestimmten Stelle im Speicher auf (vgl. 1.2.1).

Daneben hat ein Wert auch stets eine bestimmte Lebensdauer, die angibt, wie lange der Wert im Speicher existiert.
Diese Lebensdauer wird als der "scope" eines Wertes bezeichnet.
Der "scope" ist dabei abhängig von dem "mode" des Wertes sowie von der Art, wie dieser Wert im Speicher erzeugt worden ist.

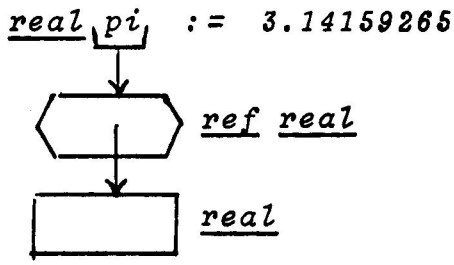
So existieren z.B. die Werte der Grund-"modes" *bool*, *char*, *int* und *real* während der ganzen Bearbeitungszeit des gesamten Programms im Speicher.
Wenn eine Adresse sich auf einem Wert bezieht, der innerhalb eines gewissen Programmteils 'lokal' (auf dem "stack") erzeugt worden ist, so existiert diese Adresse nur solange, wie dieser Programmteil abgearbeitet wird.

Beispiel:



Der *real*-Wert 3.14159265 existiert während der Bearbeitungszeit des gesamten Programms im Speicher.

Haben wir dagegen:



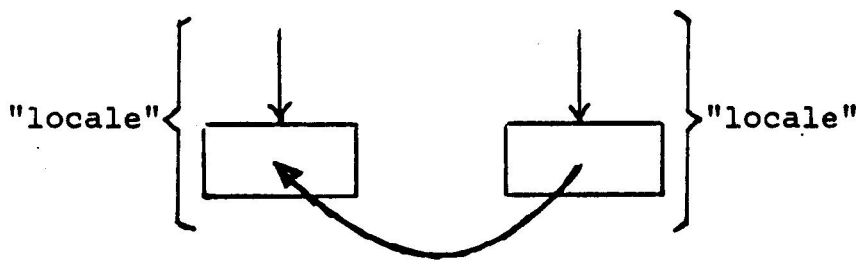
so existiert die Adresse von *pi* nur solange im Speicher, wie der Programmteil abgearbeitet wird, in dem die declaration *real pi:= 3.14159265* gültig ist.

Wir werden im Abschnitt 3.4 noch ausführlicher über den "scope" von im Speicher erzeugten Werten sprechen.

1.3. Abarbeitung eines ALGOL68-Programms

1.3.1. Wert-Transport und assignation

Wenn innerhalb des Speichers ein Wert von einer Speicherzelle in eine andere Speicherzelle 'transportiert' wird, so bedeutet dies stets, dasz ein Wert aus einem "locale" in einen anderen "locale" kopiert wird. Dies veranschaulichen wir durch folgendes Bild:



"Transport eines Wertes"

Man beachte hierbei, dasz nach dem Wert-'Transport' beide "locales" den gleichen Wert enthalten.

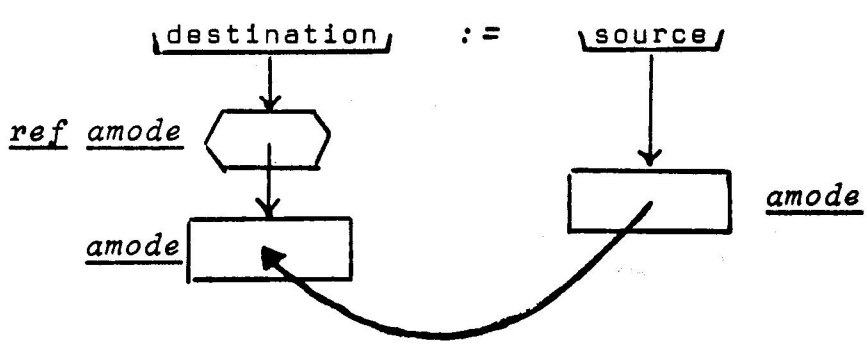
Der Wert-Transport kann als eine elementare Aktion angesehen werden. Diese elementare Aktion wird in ALGOL68 bei der "elaboration" einer assignation vorgenommen.

Die allgemeine Form einer assignation ist:

```
destination    :=    source
```

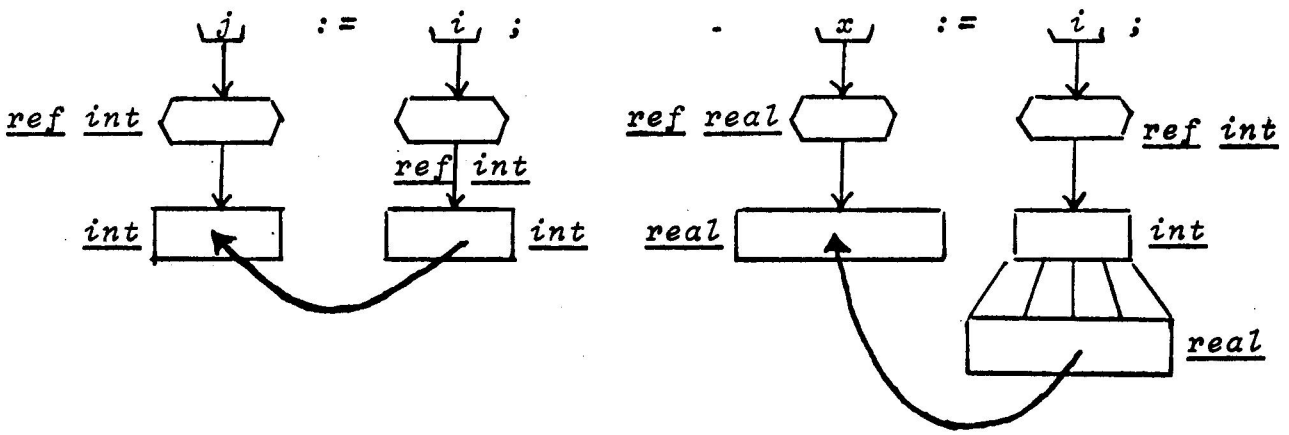
Die destination musz eine Adresse, also ein *ref amode* liefern; die source musz dann ein *amode* liefern. Wenn die destination keinen *ref amode* oder die source den von der destination geforderten *amode* nicht liefern kann, so ist die assignation syntaktisch inkorrekt (es erfolgt dann hoffentlich eine Fehlermeldung des Compilers).

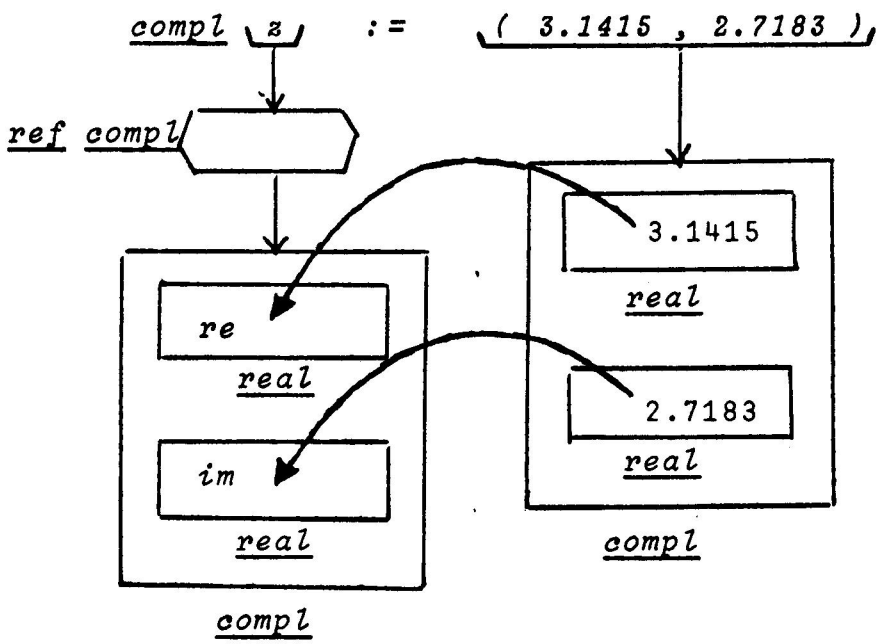
Die "elaboration" einer assignation wird durch folgendes Bild veranschaulicht:



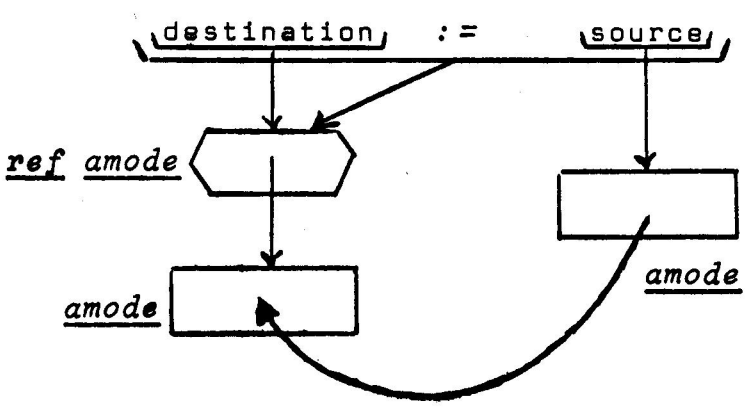
Für "*amode*" kann man jeden "mode" einsetzen. Also ist der Wert-'Transport' bei einer ALGOL68-assignation definiert für Werte von beliebigen "mode".

Beispiele:





Wichtig ist noch, dasz eine assignation selbst als externes Objekt ein internes Objekt (also einen Wert) erhält und zwar die Adresse, die ihre destination liefert (vgl. 2.2.1):



Diese assignation erhält also den *ref amode*, der von seiner destination geliefert wurde.

1.3.2. Vergleich von Werten

Wir haben in 1.2.3 gesehen, dasz zwischen bestimmten Werten (also internen Objekten) die Relationen "to be equivalent to" und "to be smaller than" definiert sein können.

Für den allgemeinen Vergleich von Werten können diese beiden Relationen als 'Axiome' angesehen werden; denn ausgehend von ihnen werden in der "standard-prelude" (vgl. 1.3.4) alle Vergleichsoperatoren:

$$= \quad \neq \quad < \quad > \quad \leq \quad \geq$$

definiert.

Es wird also 'axiomatisch' vorausgesetzt, dasz man:

a) zu jedem *int* stets einen *real* finden kann, der gleich ("equivalent") diesem *int* ist,

b) von je zwei *int*s und von je zwei *real*s stets entscheiden kann, welcher von beiden der kleinere ist.

1.3.3. Routinen und Operatoren

Eine Routine wird bestimmt durch ein routine-text (vgl. 3.2.2) und ist selbst ein Wert (ist also Bestandteil eines "locale"): ein Stück Objektcode.

Wir werden sehen, dasz identifiers und Operatoren Routinen besitzen können, dasz auch Adressen von Routinen eine Rolle spielen können, dasz man Routinen im Speicher 'transportieren' (zuweisen) kann, etc. Man kann also eine Routine wie jeden anderen Wert betrachten.

Eine Routine kann Parameter haben (jedoch auch ohne Parameter glücklich sein), und sie liefert stets einen Wert (der aber auch *void* sein kann).

Zunächst interessiert uns die Tatsache, dasz ein Operator eine Routine erhält. Es gibt dann zwei Möglichkeiten:

1) monadischer Operator: die Routine hat einen Parameter, den rechten Operanden ,

2) dyadischer Operator: die Routine hat zwei Parameter, einen linken und einen rechten Operanden .

Darüberhinaus hat ein Operator noch eine Priorität, die eine implizite Klammerung bewirkt:

$$a + b \times c$$

bedeutet:

$$a + (b \times c)$$

da × eine höhere Priorität hat als +.

1.3.4. Standard-Prozeduren und Standard-Operatoren

Alle gebrächlichen Operatoren und einige Prozeduren (die meisten sind vom "mode" *proc(real)real*) sind in ihrer Bedeutung in de standard-prelude definiert.

Wir geben hier einen vorläufigen Ausschnitt daraus (es gibt weitaus mehr Standard-Operatorenals hier angegeben).

Standard-Prozeduren

identifier	argument	result
sqrt	*real*	*real*
exp		
ln		
cos		
arccos		
sin		
arcsin		
tan		
arctan		
read	*ref amode*	*void*
print	*amode*	*void*

Hierbei ist *amode* jeder Wert, der gelesen oder ausgedruckt werden kann.

dyadische Operatoren

operator	priority	left operand	right operand	result
+	6	*int*	*int*	*int*
-	6	*int*	*real*	*real*
		real	*int*	*real*
×	7	*real*	*real*	*real*
/	7	*int*	*int*	*real*
		int	*real*	*real*
		real	*int*	*real*
		real	*real*	*real*
÷ *over*	7	*int*	*int*	*int*
mod	7			
↑ ×× *up*	8	*int*	*int*	*int*
		real	*int*	*real*
and	3	*bool*	*bool*	*bool*
or	2			
= *eq*	4	*bool*	*bool*	*bool*
≠ *ne* /=	4	*char*	*char*	*bool*
		int	*int*	*bool*
		int	*real*	*bool*
		real	*int*	*bool*
		real	*real*	*bool*
< *lt*	5	*char*	*char*	*bool*
> *gt*	5	*int*	*int*	*bool*
		int	*real*	*bool*
≤ *le* <=	5	*real*	*int*	*bool*
≥ *ge* >=	5	*real*	*real*	*bool*

monadische Operatoren

operator	priority	operand	result
not	10	*bool*	*bool*
+ -	10 10	*int* *real*	*int* *real*
abs	10	*bool* *char* *int* *real*	*int* *int* *int* *real*
odd	10	*int*	*bool*
sign	10	*int* *real*	*int* *int*
round *entier*	10 10	*real*	*int*
repr	10	*int*	*char*

Operatoren zur Abfrage der Grenzen von "multiple values"

operator	priority	left operand	right operand	result
lwb *upb*	10 10		()*amode*	*int*
lwb *upb*	8 8	*int*	(,,)*amode*	*int*

Operatoren für *compl*s

operator	priority	left operand	right operand	result
+	6	*compl*	*compl*	*compl*
-	6	*compl*	*real*	*compl*
×	7	*real*	*compl*	*compl*
/	7	*compl*	*int*	*compl*
		int	*compl*	*compl*
+	10		*compl*	*compl*
-	10			
conj				
re	10		*compl*	*real*
im	10			
abs	10			
i ⊥	9	*real*	*real*	*compl*
↑ ×× *up*	8	*compl*	*int*	*compl*
= *eq*	4	*compl*	*compl*	*bool*
≠ *ne* /=	4	*compl*	*real*	*bool*
		real	*compl*	*bool*
		compl	*int*	*bool*
		int	*compl*	*bool*

Operatoren für *strings* und *chars*

operator	priority	left operand	right operand	result
+	6	*string* *string* *char*	*string* *char* *string*	*string* *string* *string*
= *eq* ≠ *ne* /= < *lt* > *gt* ≤ *le* <= ≥ *ge* >=	4 4 5 5 5 5	*string* *string* *char*	*string* *char* *string*	*bool* *bool* *bool*

Zuweisende Operatoren

operator	priority	left operand	right operand	result
plusab +:= *minusab* -:= *timesab* ×:= *divab* /:=	1 1 1 1	*ref int* *ref real* *ref real* *ref compl* *ref compl* *ref compl*	*int* *int* *real* *int* *real* *compl*	*ref int* *ref real* *ref real* *ref compl* *ref compl* *ref compl*
plusab +:=	1	*ref string* *ref string*	*char* *string*	*ref string* *ref string*
prusab +=:	1	*char* *string*	*ref string* *ref string*	*ref string* *ref string*

1.3.5. Serielle und kollaterale Abarbeitung

Die Abarbeitung von externen Objekten durch die ALGOL68-Maschine ist nicht immer von deren textlicher Reihenfolge bestimmt. Die ALGOL68-Maschine hat durchaus die Möglichkeit, sich in mehrere Prozessoren zu trennen, die dann unabhängig voneinander verschiedene externe Objekte abarbeiten. Wir werden allmählich erkennen, dasz die ALGOL68-Maschine sehr viele Möglichkeiten bei der Abarbeitung von externen Objekten hat.

Ausdrücke in ALGOL68, die zeitlich nacheinander ausgeführt werden sollen, werden voneinander durch Semikolons ";" getrennt. Dieses Semikolon heiszt in ALGOL68: go-on-token.

Ausdrücke in ALGOL68, bei denen die zeitliche Ausführungsfolge (vom Standpunkt des Programmierers aus gesehen) keine Rolle spielt, kann man durch Kommas "," trennen. Dieses Komma heiszt in ALGOL68: and-also-token.

"Serial elaboration" (serielle Abarbeitung)
bedeutet:

> Die zeitliche Ausführungsfolge entspricht völlig der textlichen Reihenfolge (abgesehen von einem Sprung). Die "elaboration" geschieht schrittweise von go-on-token zu go-on-token.

Beispiel:

begin real x; read (x); x:= x↑2×sqrt (x); print (x) end

Zuerst	wird eine Speicherstelle für *x* reserviert durch Abarbeitung von *real x* ;
danach	wird *read (x)* abgearbeitet ;
danach	wird die assignation *x:= x↑2×sqrt (x)* abgearbeitet ;
und danach	wird *print (x)* abgearbeitet .

Die vier Komponenten des Programms:

real x ; *read (x)* ; *x:= x↑2×sqrt (x)* und *print (x)*

werden also in ihrer textlichen Reihenfolge (seriell) abgearbeitet: "serial elaboration".

"Collateral elaboration" (kollaterale Abarbeitung) bedeutet:

> Die zeitliche Ausführungsfolge ist in keiner Weise von der textlichen Reihenfolge her abzuleiten.

Beispiel:

> *begin* $u := x\,(y+z)+a$, $v := x\,(y+z)-a$, $w := x\,(y+z)\times a$ *end*

Hierbei sind drei assignations durch and-also-tokens getrennt; diese drei assignations werden 'kollateral' ("collateral") ausgeführt, d.h: man weisz nicht, in welcher zeitlichen Reihenfolge der Computer diese zusammengesetzten Ausdrucke bearbeitet: vielleicht fängt er mit der zweiten assignation an, bearbeitet dann die dritte und zum Schlusz die erste assignation, oder
er führt die Bearbeitung in irgendeiner anderen denkbaren Reihenfolge aus, oder
er berechnet $x\uparrow(y+z)$ nur einmal und führt danach in irgendeiner zeitlichen Reihenfolge die drei Zuweisungen aus; man kann sich durchaus einen 'schlauen' Compiler vorstellen, der eine derartige Optimierung vornimmt.
Man kann sich bei "collateral elaboration" am besten das folgende Bild vor Augen führen:

Die ALGOL68-Maschine trennt sich in (in unserem Beispiel) drei Prozessoren, wobei zunächst jeder für sich eine der drei assignations bearbeitet, sodann jeder wartet, bis alle drei fertig sind.
Dabei hat der Programmierer immer folgendes zu beachten:

> Bei der "collateral-elaboration" sollten die kollateralen Komponenten keine Nebeneffekte aufeinander ausüben; denn treten solche Nebeneffekte auf, so sind diese völlig undefiniert.

Beispiel:

> $n := 0;\quad print\,((u := n+1\ ,\ v := n+2\ ,\ w := n+3))$

Es ist sehr wichtig zu verstehen, was hier passiert. Es sind nämlich zwei Dinge, die man auseinander halten musz:

1) Die Prozedur *print* erwartet stets eine Liste von Werten, z.B. vom "mode" *bool*, *char*, *int* oder *real*. Dabei werden diese Werte stets in ihrer, in der Liste auftretenden Reihenfolge (also: in ihrer textlichen Reihenfolge) ausgedruckt.

Für unser Beispiel bedeutet dies:

$$print\ (\ \overbrace{\underbrace{u:=n+1}_{\text{Wert1}}\ ,\ \underbrace{v:=n+2}_{\text{Wert2}}\ ,\ \underbrace{w:=n+3}_{\text{Wert3}}}^{\text{Liste}}\ )$$

Es werden also 'Wert1', 'Wert2' und 'Wert3' in dieser Reihenfolge ausgedruckt.

2) Bevor jedoch ausgedruckt werden kann, müssen die Werte berechnet werden, und diese Berechnung geschieht (wegen der and-also-tokens) kollateral.

In dem angegebenen Beispiel kann nichts 'Schlimmes' passieren, da die kollateralen Komponenten keine Nebeneffekte aufeinander ausüben. Ausgedruckt wird stets in dieser Reihenfolge:

1	2	3

Betrachten wir dagegen folgendes Beispiel:

n:= 0 ; print ((n:= n+1 , n:= n+2 , n:= n+3))

Jetzt üben alle drei kollateralen Komponenten Nebeneffekte aufeinander aus: sie verändern alle drei den Wert von *n*.
Es kann ausgedruckt werden:

1	3	6
6	2	5
4	6	3
1	2	3
1	2	5

etc.

Nochmals: Vermeide Nebeneffekte in kollateralen Komponenten.

Dennoch, das Konzept der Kollateralität ist von groszer begrifflicher Bedeutung:

a) Wie später noch ersichtlich wird, ist die Kollateralität der völlig unsynchronisierte (unkontrollierte) Fall des "parallel-processing" (vgl. 8.5).

b) Die Kollateralität zwingt den Programmierer, Nebeneffekte zu vermeiden (da sonst ---, siehe obiges Beispiel). Anders gesagt, will der Programmierer Nebeneffekte erzielen, so verwende er die "serial-elaboration"; denn dann hat er auch Kontrolle über diese Nebeneffekte.

c) Weil Nebeneffekte keine 'legale' Rolle spielen, besteht für den Compiler die Möglichkeit, einen effizienten Code für die Ausführung ("execution") zu liefern.

In den folgenden Beispielen hat der Compiler solche Optimierungsmöglichkeiten:

```
y := axb + sqrt (axb) ;

real u:= x↑(y+z)+a , real v:= x↑(y+z)-a ;
```

1.4. Programm-Beispiele, verschiedene Schreibweisen

Folgende Beispiele illustrieren einiges von dem, was in den vorgehenden Abschnitte behandelt wurde. Man betrachte sie in erster Instanz als Beispiele zum Lesen; man braucht an dieser Stelle noch nicht zu wissen, wie man solche Programme aufstellt.

Im zweiten Kapitel wird die Satzbau eingehend behandelt, und vieles, das beim ersten Lesen vielleicht noch fremd und überraschend ist, wird dann in dem logischen Zusammenhang einleuchtend werden.

Nach diesem zweiten Kapitel sollte man die Beispiele noch einmal betrachten.

1.4.1. Komprimierte Schreibweisen

In ALGOL68 gibt es alternative Schreibweisen für viele Sprachkonstruktionen. Der Zweck ist meistens: Abkürzung und/oder gröszere Übersichtlichkeit. Man gewöhnt sich ziemlich schnell daran.

So kann man statt

begin serial-clause *end*

immer auch schreiben:

(serial-clause)

(für eine serial-clause vgl. 2.3 und auch 1.3.5)

Auch die in 1.3.5 besprochenen Sprachkonstruktionen führen zu bemerkenswerten Abkürzungen des Programmtextes, die unter Umständen den Compiler helfen können den Objektcode zu optimieren. So wird Beispiel {1} (aus 1.1.1) in komprimierter Form:

```
{1'}    ( real x, y, sum, squm;
          read((x,y));
          (sum:=x+y , squm:=x↑2+y↑2);
          print((sum/2 , sqrt(2×squm-sum 2)/2))
        )
```

Normalerweise werden wir aber *begin* und *end* schreiben wenn die serial-clause auch declarations enthält.

1.4.2. Label, *goto* und conditional-clause

Ein Kernthema der Algorithmiek ist die Wiederholung. Die weithin primitivste Methode, sie in einem Programm zum Ausdruck zu bringen ist die folgende:

> markiere die Stelle im Programmtext, an der die Schleife beginnt, mit einem label,
> schreibe einen jump (also: *goto* label-identifier) dort, wo die Schleife endet.

Die Entscheidung, ob ein jump ausgeführt werden soll oder nicht, wird of von einer 'condition' abhängig gemacht. Man kann dann eine conditional-clause verwenden:

if 'condition' *then* jump *fi*

Eine vollständige Darstellung der conditional-clause findet man in 2.5.1.
Einige einfache Formen einer conditional-clause werden wir schon in den folgenden Beispielen betrachten, man wird sie leicht verstehen.

```
{2}     begin int k, n;
            real next, sum, squm;
             read(n);
             sum:=squm:=0;
             k:=1;
        do:  read(next);
             (sum:=sum+next , squm:=squm+next↑2 ,
              k:=k+1);
             if k≤n then goto do fi;
             print((sum/n , sqrt(n×squm-sum↑2)/n))
        end
```

Das goto-token *goto* (man darf sogar auch "*go to*" schreiben) darf man weglassen und statt

if 'condition' *then* jump *fi*

darf man

('condition' | jump)

schreiben.

Variable darf man in ihrer declaration gleich initialisieren. Zwischen declarations darf man statements (vgl. 2.10) schreiben, aber keinen label.

Benutzen wir auch noch die zuweisenden Operatoren (vgl. 1.3.4), so wird Beispiel {2} zu:

```
{2'}    begin  int k, n; read(n);
               real next, sum:=0, squm:=0;
                k:=1;
        do:     read(next);
                (sum plusab next , squm plusab next↑2 ,
                   k plusab 1);
                (k≤n | do);
                print((sum/n , sqrt(n×squm-sum↑2)/n))
        end
```

1.4.3. Loop-clauses

Eine loop-clause verwendet den label und den jump völlig implizit:

```
{3}     begin  int n; read(n);
               real next, sum:=0, squm:=0;
                int k:=1;
              while k≤n
                 do read(next);
                    (sum plusab next , squm plusab next↑2 ,
                       k plusab 1)
                 od;
                print((sum/n , sqrt(n×squm-sum↑2)/n))
        end
```

Man beachte, dasz der zu wiederholende Programmteil stets in *do* und *od* eingeschlossen wird:

do serial-clause *od*

Eine andere Fassung der loop-clause verwendet sogar auch die Lauf-Variable *k* implizit (wenn sie nicht in der Schleife benötigt wird):

```
{4}     begin  int n; read(n);
               real next, sum:=0, squm:=0;
                 to n
                 do read(next);
                    (sum plusab next , squm plusab next↑2).
                 od;
                print((sum/n , sqrt(n×squm-sum↑2)/n))
        end
```

Die Schleife *do* serial-clause *od* kann man auf verschiedenen Weisen steuern (vgl. auch {5} und {7}).
Für eine vollständige Darstellung der loop-clause vergleiche 2.7.1 und 2.7.2.

1.4.4. Reihen ("multiple values")

Eine Reihe ist ein μ-dimensionales Gebilde (μ=1,2,3,---) von Werten gleichen "modes" (also ein "*array*", dieser Term wird in ALGOL68 jedoch nicht mehr standardmäszig benutzt, man braucht ihn auch gar nicht).

Bei ihrer declaration werden die Grenzen der Reihe festgelegt und meistens auch für ihre Lebensdauer fixiert. Es gibt aber Reihen (wie z.B. der *string*, vgl. 1.1.3), deren Grenzen auch nach der declaration in einer assignation noch geändert werden dürfen.

Mit vielleicht einiger Mühe wird man folgendes Beispiel verstehen:

```
{5}     begin  int n; read(n);
                if n≤1 then print("empty row"); goto out fi;
              (1:n)real row; read(row);
              real sum:=row(1), squm:=row(1)↑2;
            string gr:= "I am the greatest",
                   sm:= "I am the smallest";
               int imin:=1, imax:=1;
               for i from 2 to n
                do (sum plusab row(i) , squm plusab row(i)↑2 ,
                    if row(i)<row(imin) then imin:=i fi ,
                    if row(i)>row(imax) then imax:=i fi
                od;
               int h, k; string s1, s2;
                if imin<imax then h:=imin; k:=imax; s1:=sm; s2:=gr
                             else h:=imax; k:=imin; s1:=gr; s2:=sm
                fi;
               print((row(1:h-1)    , newline ,
                      row(h) , s1   , newline ,
                      row(h+1:k-1)  , newline ,
                      row(k) , s2   , newline ,
                      row(k+1:n)    , newline , newline ,
                      "mean:      " , sum/n   , newline ,
                      "stand-dev:"  , sqrt(n×squm-sum↑2)/n
                    ));
          out: skip          comment skip is the 'dummy-statement' co
     end
```

1.4.5. Strukturen ("structured values")

Eine Reihe ist ein dynamisches Gebilde: seine Grenzen werden bei jeder Abarbeitung seiner declaration auf's neue berechnet und auch die Adressen der ausgewählten Elemente oder Teilreihen werden erst während der Ausführung des Programms (also während der "execution") festgestellt.
Eine Struktur dagegen ist ein statisches Gebilde: sein Umfang, wie auch die Auswahl seiner Komponenten (seiner fields) sind schon dem Compiler bekannt.

Der interessanteste Unterschied zwischen einer Reihe und einer Struktur besteht aber darin, dasz die Elemente einer Reihe alle vom gleichen "mode" sein müssen, die fields einer Struktur dagegen können sehr wohl von verschiedenem "mode" sein, und gerade das hat weitgehende Folgen (vgl. 6.2).

Ohne viel Mühe wird man folgendes Beispiel verstehen:

```
{6}     begin print(("new-capitalistic Xmas-gratification" ,
                 newline , newline));
        mode employee = struct ( int entrance year ,
                                 real salary ,
                               string name          );
        employee director, worker;
        int current year; read(current year);
        read((director , worker));
        while name of worker ≠ name of director
        do print(( newline ,
                   name of worker , newline ,
                   "$" , space , space ,
                   (current year + 1
                    - entrance year of worker) ×
                   (salary of director - salary of worker)/100
                 ));
           read(worker)
        od;
        print((newline , newline , "end of Xmas-joke"))
    end
```

1.4.6. Routinen ("routines")

Im folgenden Beispiel wird eine Prozedur *compute* definiert, die die Berechnung des Beispiels {5} mit irgendeiner (eindimensionalen) Reihe (formal-parameter) vornimmt, somit also die Indizes des kleinsten und grösten Elementes sowie den Mittelwert und die Standardabweichung als Ergebnis liefert. Die Prozedur sei definiert in einer Umgebung, in der die mode-declaration:

```
mode quadr = struct ( int smind , grind ,
                      real mean  , standev )
```

gegeben ist.
Der "mode" der "routine", die *compute* erhält, ist dann:

```
proc(()real)quadr
```

also: eine orthogonale Kombination von Grund-"modes".
Man beachte auch die Verwendung der Operatoren *lwb* und *upb* (vgl. 1.3.4).

```
{7}     proc compute = (()real r) quadr:
                begin  int m:= lwb r, n:= upb r;
                       real sum:= r(m), squm:= r(m)↑2;
                        int h:=1, k:=1;
                        for i from m+1 to n
                         do ( sum plusab r(i) ,
                              squm plusab r(i)↑2 ,
                               if r(i)<r(h) then h:=i fi ,
                               if r(i)>r(k) then k:=i fi )
                         od;
              result: ( h , k ,
                        sum/n , sqrt(n×squm-sum↑2)/n )
                end ;
```

Rechts von dem is-defined-as-token "=" steht der routine-text, der hier aus einer closed-serial-clause besteht. Das letzte in ihr (dem 'Zier'-label *result:* folgend) liefert den Wert der "routine": ein structure-display, hier ein *quadr*.
Zur vollständigen Darstellung von Prozeduren vergleiche 3.2 und 3.3; vergleiche auch 2.1.2 usw. für unit, structure-display etc.

Wie man die Prozedur *compute* aufruft, sieht man im folgenden Beispiel, das (ungefähr) das gleiche leistet wie {5}:

```
{5'}    begin  int n; read(n);
                if n≥1
              then (1:n)real row; read(row);
                   quadr statistics := compute(row) ;
                   int imin:= smind of statistics ,
                       imax:= grind of statistics ;
                   print(( row , newline , newline , newline ,
                           "row(",imin,")=",row(imin),
                           "I am the smallest" , newline ,
                           "row(",imax,")=",row(imax),
                           "I am the greatest" , newline ,
                            newline ,
                           "mean:      " , mean of statistics ,
                            newline ,
                           "stand-dev:" , standev of statistics
                         ))
              else print("empty row")
                fi
        end
```

1.4.7. Verschiedene Darstellungsformen

Ein Programmtext in ALGOL68 besteht aus einer Folge von syntaktischen Symbolen, indicators und denotations. Die Tastatur der meisten Eingabe-Apparaten (Stanzmaschinen für Lochkarten oder Lochstreifen usw.) ist aber beschränkt. Die einfachste Tastaturen umfassen nicht mehr als 48 Zeichen (1 Alphabet *A*,*B*,*C*,---,*X*,*Y*,*Z*, die Ziffer *0*,*1*,*2*,---,*9* und noch 12 der meist benötigten Zeichen); mit den moderneren Tastaturen kann man schon 64 oder sogar 96 Zeichen darstellen (2 Alphabets *A*,*B*,*C*,---,*X*,*Y*,*Z*,*a*,*b*,*c*,---,*x*,*y*,*z*, die Ziffer *0*,*1*,*2*,---,*9* und weiter noch 34 andere Zeichen). Wie schon erwähnt in 0.3 unterscheiden wir zwei Alphabets:

a,*b*,*c*,*d*,*e*,*f*,*g*,*h*,*i*,*j*,*k*,*l*,*m*,*n*,*o*,*p*,*q*,*r*,*s*,*t*,*u*,*v*,*w*,*x*,*y*,*z*
(für identifiers)

und

<u>*a*</u>,<u>*b*</u>,<u>*c*</u>,<u>*d*</u>,<u>*e*</u>,<u>*f*</u>,<u>*g*</u>,<u>*h*</u>,<u>*i*</u>,<u>*j*</u>,<u>*k*</u>,<u>*l*</u>,<u>*m*</u>,<u>*n*</u>,<u>*o*</u>,<u>*p*</u>,<u>*q*</u>,<u>*r*</u>,<u>*s*</u>,<u>*t*</u>,<u>*u*</u>,<u>*v*</u>,<u>*w*</u>,<u>*x*</u>,<u>*y*</u>,<u>*z*</u>
(das sogenannte bold-alphabet für indications und besondere Symbolen und Operatoren).

Hat man aber nicht mehr als 1 Alphabet zur Verfügung, so musz man Folgen von bold Buchstaben unterscheiden von identifiers; hierzu gibt es verschiedenen Möglichkeiten:

a) man verwendet den Apostroph "'" um Beginn und Ende einer Folge von bold Buchstaben zu markieren,

b) man verwendet den Apostroph "'" als "boldface-shift": er markiert dann den Beginn der Folge von bold Buchstaben; die "boldface-shift" beeinfluszt alle Buchstaben bis ein Zeichen (inklusive "blank") ungleich Buchstab oder Ziffer folgt.

Es wird klar sein dasz die Conventionen a) und b) nicht durcheinander verwendet werden können.

Hat man zwei Alphabets zur verfügung, so kann man z.B. die Grosz-Buchstaben verwenden für die bold Buchstaben und die Kleinbuchstaben für die identifiers (oder, so man will, gerade umgekehrt).

ALGOL68 braucht, neben die 'normale' Klammer "(" und ")", auch noch die Klammer "[" und "]". Hat man die rechte Klammer nicht zur Verfügung, so kann man "(/" und "/)" verwenden.

Im folgenden geben wir verschiedenen Darstellungen des Programms:

```
begin  int n;  read(n);
      (1:n)real row;  read(row);
       int imax:=1;  real sum:=row(1);
       for i from 2 to n
        do if row(i)>row(imax) then imax:=i fi;
           sum plusab row(i)
        od;
        print(( row(imax) , "I am the greatest" ,
                sum
              ))
end
```

Wenn man zwei Alphabets zur Verfügung hat:

```
BEGIN  INT n;  read(n);
      (1:n)REAL row;  read(row);
       INT imax:=1;  REAL sum:= row(1);
       FOR i FROM 2 TO n
        DO IF row(i)>row(imax) THEN imax:=i FI;
           sum PLUSAB row(i)
        OD;
        print(( row(imax) , "I am the greatest" ,
                sum
              ))
END
```

oder umgekehrt:

```
begin   int N;  READ(N);
       (1:N)real ROW;  READ(ROW);
        int IMAX:=1;  real SUM:=ROW(1);
        for I from 2 to N
         do if ROW(I)>ROW(IMAX) then IMAX:=I fi;
            SUM plusab ROW(I)
         od;
         PRINT(( ROW(IMAX) , "I am the greatest" ,
                 SUM
               ))
end
```

Wenn man nur 1 Alphabet zur Verfügung hat, und nicht die Zeichen: "(", ")" und ">", dann mit a):

```
'BEGIN' 'INT' N;  READ(N);
        (/1:N/)'REAL' ROW;  READ(ROW);
        'INT' IMAX:=1;  'REAL' SUM:=ROW(/1/);
        'FOR' I 'FROM' 2 'TO' N
         'DO' 'IF' ROW(/I/)'GT'ROW(/IMAX/)
            'THEN' IMAX:=I
              'FI';
              SUM 'PLUSAB' ROW(/I/)
         'OD';
         PRINT((ROW(/IMAX/) , "I AM THE GREATEST" ,
                SUM
              ))
'END'
```

oder mit b):

```
'BEGIN  'INT N;  READ(N);
       (/1:N/)'REAL ROW;  READ(ROW);
        'INT IMAX:=1;  'REAL SUM:=ROW(/1/);
        'FOR I 'FROM 2 'TO N
         'DO 'IF ROW(/I/)'GT ROW(/IMAX/)
           'THEN IMAX:=I
             'FI;
             SUM 'PLUSAB ROW(/I/)
         'OD;
         PRINT((ROW(/IMAX/) , "I AM THE GREATEST" ,
                SUM
              ))
'END
```

1.4.8. Die particular-prelude dieses Buches

Wenn in diesem Buch identifiers und indications auftreten ohne in einer declaration (die auch in der standard-prelude auftreten kann) definiert zu sein, so nehmen wir stets an, dasz sie wie folgt definiert sind:

{8}
```
mode vector = (1:n)real ,
mode matrix = (1:n,1:n)real ;

mode man   = struct(string name, int year,
                    ref woman wife) ,
mode woman = struct(string name, int year,
                    ref man   husband ,
                    ref flex (1:0)human child) ,
mode human = union(man,woman) ;

mode number = union(int,real,compl) ,
mode birc   = union(bool,number);

mode fun    = proc(real)real ;
```

{9}
```
bool p, q, b1, b2, b3;
char c, symbol;
int  h, i, j, k, m, n;
real a, b, u, v, w, x, y;

compl alfa, beta, gamma, delta, epsilon,
      z;

vector row, x1, y1;
matrix rowrow, x2, y2;

string string, text, symrow, s, s1, s2;

man   romeo;
woman juliet;
human uli;

fun   f, g;
```

2. STRUKTUR DER SPRACHE

2.1 Einfache Syntax

2.1.1 Einige Bemerkungen zum Satzbau
2.1.2 Phrases

2.2 Der Wert einer phrase

2.2.1 Der Wert eines unit
2.2.2 "apriori"- und "aposteriori-mode"

2.3 serial-clause

2.3.1 Der äuszere Aufbau einer serial-clause
2.3.2 Der Wert einer serial-clause
2.3.3 Die closed-serial-clause und ihr Wert

2.4 Deklaration und Identifizierung

2.4.1 Deklarationen
2.4.2 Identifizierung

2.5 Die conditional-clause

2.5.1 Der äuszere Aufbau einer conditional-clause
2.5.2 Der Wert einer conditional-clause
2.5.3 Schachtelung von conditional-clauses

2.6 Die case-clause

2.6.1 Der äuszere Aufbau einer case-clause
2.6.2 Der Wert einer case-clause

2.7 Die loop-clause

2.7.1 Der äuszere Aufbau einer loop-clause
2.7.2 Verschiedene Formen einer loop-clause

2.8 Die closed-collateral-clause

2.8.1 Der äuszere Aufbau einer closed-collateral-clause

2.8.2 Der Wert einer closed-collateral-clause

2.9 Der completer

2.10 Schluszbemerkung zum Satzbau

---

2. STRUKTUR DER SPRACHE

2.1. Einfache Syntax

==========

2.1.1. Einige Bemerkungen zum Satzbau

Die Syntax von ALGOL68, wie im Report angegeben, ist alles andere als einfach. Die Hauptursache hierfür liegt darin, dasz man eine Sprache konstruiert hat, bei welcher der Programmierer sich in erster Linie von seiner semantischen Intuition leiten lassen kann. In ALGOL68 ist jede syntaktische Konstruktion der Träger eines einzigen bestimmten semantischen Inhalts. Die orthogonale Struktur der Sprache hat zur Folge, dasz verschiedene syntaktische Konstruktionen Träger von voneinander unabhängigen semantischen Inhalten sind, die überdies beliebig miteinander kombiniert werden dürfen. Wenn man also die Bedeutung (die Semantik) einer Sprachkonstruktion gut verstanden hat, kennt man fast zwangsläufig auch seine syntaktische Gestalt und Kombinationsmöglichkeiten.
Dies bedeutet jedoch nicht, dasz der Programmierer gänzlich ohne Kenntnis der Syntax seine Programme hinschreiben kann: es gibt unvermeidlich formale Verabredungen in Schreibweise und Symbolauswahl, die man wissen musz. Man kann jedoch sagen, wenn ein Stück "ALGOL68-Prosa" nach kritischer Betrachtung klar, logisch und unzweideutig geschrieben worden ist, so wird es in den meisten Fällen auch syntaktisch korrekt sein.
Die ALGOL68-Syntax ist vor allem auffallend liberal: Gebots- und Verbotsregeln bestehen nur, um semantisch unlogische oder mehrdeutige Konstruktionen soweit wie möglich zu vermeiden und, falls irrtümlicherweise doch vorhanden, diese rechtzeitig (d.h. schon zur "compile-time") zu erkennen.

2.1.2. Phrases

Ein Programm in ALGOL68 besteht aus phrases, die durch go-on-tokens ("serial-elaboration") voneinander getrennt werden. Eine phrase ist dabei entweder eine declaration oder ein unit.

Um zu einem Verständnis des Satzbaues zu kommen, werden im folgenden in einem recht groben Überblick die verschiedenen Arten von phrases aufgeführt. Dabei wird alles das weggelassen, was für den praktischen Gebrauch nicht von unmittelbarem Interesse ist (vgl. jedoch Kapitel 9).
Unser Standpunkt ist dabei der folgende: die offizielle Syntax (angegeben im Report) beschreibt die Sprache für den Compilerbauer, unser grober Überblick ist dagegen ein Übersichtsplan, mit Hilfe dessen der Programmierer sich innerhalb der Sprache, ohne allzu kleinlich zu sein, zurechtfinden kann.

- phrase
  - declaration
    - mode-declaration
    - identifier-declaration
    - operation-declaration
    - priority-declaration
  - unit
    - 'closed unit'
      - enclosed-clause
        - closed-serial-clause
        - closed-collateral-clause
        - choice-clause
          - conditional-clause
          - conformity-clause
          - case-clause
      - loop-clause
    - 'open unit'
      - units, die nicht in irgendeiner Weise 'geklammert' sind, wie zum Beispiel:
      - denotation, identifier, formula, assignation, call, slice, selection, jump,
      - weitere 'open units' werden wir später noch kennenlernen

Die Wörter, die wir schreiben im Alphabet "abcdefghijklmnopqrst uvwxyz" sind NOTIONS im Sinne des Reports, das heiszt: syntaktische Begriffe, für welche in der formalen Syntax im Report Produktionsregeln gegeben sind (vgl. Kapitel 9).

Durch eine enclosed-clause als mögliches unit und so als mögliche phrase erhält man die 'Block-Struktur' in der Sprache. Eine enclosed-clause ist die Verallgemeinerung der von ALGOL60 und anderen (ALGOL-ähnlichen) Programmiersprachen her bekannten Begriffe wie 'block', 'compound-statement', 'geklammerter Ausdruck', etc.

2.2. Der Wert einer phrase

Ist eine phrase eine declaration, so liefert sie niemals einen Wert. Ist sie dagegen ein unit, so liefert sie stets einen Wert. Dabei kann der "mode" dieses Wertes auch *void* sein, dies bedeutet, der Wert wird nicht weiter betrachtet.

2.2.1. Der Wert eines units

Wir haben bereits schon einige 'open units' kennengelernt. Hier wollen wir uns im einzelnen anschauen, welche Werte sie liefern:

Der Wert

einer denotation	ist der Wert, der durch die denotation selbst gegeben ist,
eines identifier	ist der Wert, den er durch seine declaration erhalten hat,
einer formula	ist der 'berechnete' Wert,
einer assignation	ist die Adresse ihrer destination
eines call	ist der Wert, den die aufgerufene "routine" liefert,
eines slice	ist der Teil eines "multiple value", der durch die im slice angegebenen Grenzen 'ausgeschnitten' wird (oder auch, abhängig von dem Kontext, dessen Adresse),
einer selection	ist ein ausgewähltes "field" eines "structured value" (oder auch, abhängig von dem Kontext, dessen Adresse),
eines jump	betrachten wir stets als *void* .

Beispiele:

denotation *1* erhält den Wert 1; *314e-2* erhält den Wert 3.14; *true* erhält den Wert "wahr".

identifier *x* erhält eine Adresse (*ref real*), die sich auf ein *real* bezieht (vgl. 1.2.5); *row* erhält eine Adresse (*ref* ()*real*), die sich auf ein (*1:n*)*real* bezieht (vgl. 1.4.8.).

formula — *n×squm* liefert das Produkt der beiden *reals*, auf die sich die Adressen von *n* und *squm* beziehen;
n×squm - sum↑2 liefert die Differenz der Werte, die die beiden formulas *n×squm* und *sum↑2* liefern.

assignation — *squm:= x↑2+y↑2* erhält die Adresse (*ref real*) von *squm*.

call — *sqrt(n×squm-sum↑2)* liefert den Wert, den die mit dem aktuellen Parameter *n×squm-sum↑2* versehene "routine" von *sqrt* liefert (also die Quadratwurzel aus dem Wert von *n×squm-sum↑2*).

slice — *row[1]* erhält die Adresse des ersten Elements von *[1:n]real*, auf das sich die Adresse von *row* bezieht;
row[h+1:k-1] erhält die Adresse (*ref* []*real*) der durch *[h+1:k-1]* ausgeschnittenen Teilreihe.

selection — *salary of director* erhält die Adresse des zweiten "field" der Struktur (ein *employee*), auf die sich die Adresse von *director* bezieht.

jump — *goto do* erhält *void* .

Die Werte der 'closed units' werden wir in den folgenden Abschnitten eingehend behandeln. Dabei werden wir sehen, dasz ihre Werte stets durch 'open units' bestimmt werden.

Die Idee, dasz jedes unit stets einen wohl-definierten Wert (einschlieszlich *void*) liefert, ist eine der wichtigsten Neuerungen von ALGOL68. Aus dieser Idee folgt auch die grosze Flexibilität in der Schreibweise.

2.2.2. "apriori-" und "aposteriori-mode"

Was wir bisher über Werte gesagt haben, ist eine Vereinfachung der tatsächlichen Verhältnisse. Die in 2.2.1. angegebenen Werte waren stets vom "apriori-mode", somit selbst "apriori"-Werte.
Dagegen ist der "aposteriori"-Wert eines unit der vom Kontext geforderte. Den "aposteriori-mode" kann man immer vom "apriori-mode" ableiten.

Beispiel:

$$y := i \times j$$

Die "apriori"-Werte von i und j sind *ref* *int*s, also Adressen, die sich auf *int*s beziehen.
Durch den Operator x werden aber nicht die Adressen, sondern die *int*s addiert. Die "aposteriori"-Werte von i und j sind also *int*s.
Der "apriori"-Wert der formula $i \times j$ ist ein *int*, jedoch wünscht die destination y ein *real*. So ist also der "aposteriori"-Wert von $i \times j$ ein *real*.
Mit Hilfe des cast (vgl. 4.4.3.) hat man die Möglichkeit, den jeweiligen "aposteriori-mode" explizit anzugeben. Man könnte in unserem Beispiel schreiben:

$$y := real(int(i) \times int(j))$$

Soviel Mühe brauchen wir uns aber nicht zu machen, denn der Compiler 'versteht'

$$y := \quad i \times \quad j$$

auch ohne explizite Angabe der "aposteriori-modes".
Erweisz sogar, was sich dabei alles im Dunkeln abspielt, um von den "apriori-modes" zu den "aposteriori-modes" zu gelangen (in unserem Beispiel: "dereferencing" und "widening"). Das, was dort im Dunkeln passiert, sind die "coercions" (Anpassungen oder, noch bildhafter, Vergewaltigungen). Hierüber wird später noch viel und eingehend gesprochen werden (vgl. Kapitel 4).
Bei dem Entwurf von ALGOL68 hat man versucht, diese "coercions" so zu definieren, dasz sie automatisch dasjenige tun, was der Programmierer intuitiv erwartet.
So erscheint zunächst die Anwendung des cast überflüssig zu sein. Dies ist jedoch nicht immer der Fall. Es gibt Umstände, wo der Programmierer mit dem, was die "coercions" automatisch tun, nicht

einverstanden ist.

Für solche 'Querulanten' bietet ALGOL68 durch den cast die Möglichkeit, explizit zu sagen, was sie stattdessen wollen.

Nicht repräsentativ, aber dennoch einleuchtend ist folgendes Beispiel:

Wenn man schreibt

$$y := i \times j \quad ,$$

so liest der Compiler das als

$$y := \underline{real}\,(\underline{int}\,(i) \times \underline{int}\,(j))$$

In den meisten Implementationen wird der Objektcode für die Multiplikation von ganzen Zahlen ein anderer sein als für die Multiplikation von Gleitpunktzahlen. Es könnte nun sein, dasz der Programmierer der Gleitpunktmultiplikation den Vorzug gibt (z.B. wenn häufig "overflow" bei der Multiplikation von ganzen Zahlen auftreten kann). Er kann dann explizit schreiben:

$$y := \underline{real}\,(i) \times \underline{real}\,(j)$$

Der Compiler erweitert ("widening") dann die beiden Operanden zu *reals* und erzeugt dementsprechend den Code für die Multiplikation von Gleitpunktzahlen.

2.3. Serial-clauses

2.3.1. Der äuszere Aufbau einer serial-clause

Eine serial-clause besteht aus einer oder mehr phrases. Enthält sie mehr als eine phrase, so werden diese voneinander durch go-on-tokens ";" getrennt (vgl. jedoch 2.9), d.h. die phrases werden also in ihrer textlichen Reihenfolge nacheinander ausgeführt (abgesehen von einem Sprung): serielle Abarbeitung, vgl. 1.3.5.

In einer serial-clause dürfen declarations und units gemischt auftreten (eine phrase kann entweder eine declaration oder ein unit sein). Allerdings, die letzte phrase in einer serial-clause darf niemals eine declaration sein.
Der Teil einer serial-clause, in dem declarations auftreten (bis einschlieszlich der letzten declaration in der serial-clause), wollen wir 'declaration-prelude' nennen. Den restlichen Teil einer serial-clause nennen wir 'unit-sequence'.

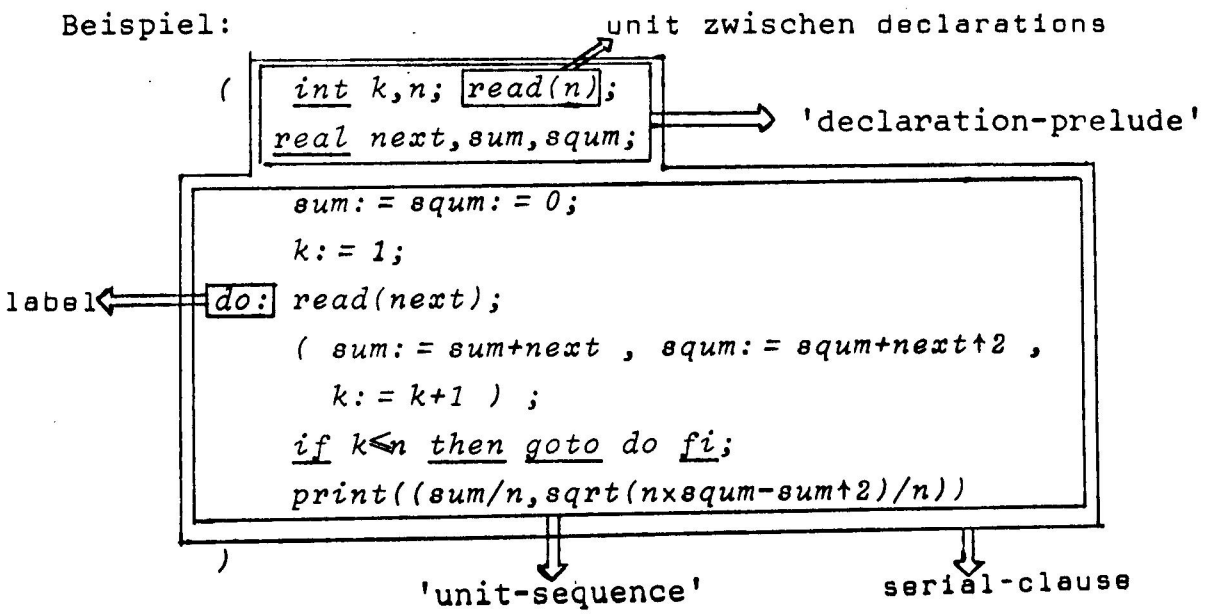
Es gilt dann:

a) die 'declaration-prelude' darf leer sein; es braucht also ein serial-clause nicht unbedingt declarations zu enthalten;

b) die 'unit-sequence' musz mindestens ein unit enthalten, darf also nicht leer sein.

Ein label darf nur in der 'unit-sequence' vokommen, anders ausgedrückt: man darf ein label erst dann vor ein unit schreiben, wenn man sicher ist, dasz keine declarations mehr folgen.

Beispiel:



2.3.2. Der Wert einer serial-clause

Eine serial-clause hat immer einen Wert (einschliesslich *void*). Dieser Wert ist der Wert des letzten unit innerhalb der serial-clause (vgl. jedoch 2.9).

Beispiel:

```
begin   int n; read(n);
        real next,sd, sum:= 0, squm:= 0;
         to n
         do read(next);
            ( sum plusab next , squm plusab next↑2 )
         od;
         sd:= sqrt(n×squm-sum↑2)/n;
         print((sum/n,sd))
end
```

Zwischen *begin* und *end* steht eine serial-clause: das letzte unit in ihr ist *print((sum/n,sd))*.
Diese Standard-Prozedur *print* liefert als Wert ein *void*, damit liefert die gesamte serial-clause ebenfalls den Wert *void*.
Will man dagegen, dasz die serial-clause etwas anderes liefert, z.B. den Wert von *sd*, so füge man noch den identifier *sd* als einfaches unit hinzu:

```
(   int n; read(n);
   real next,sd, sum:= 0, squm:= 0;
    to n
    do read(next);
       ( sum plusab next , squm plusab next↑2 )
    od;
    sd:= sqrt(n×squm-sum↑2)/n;
    print((sum/n,sd));
    sd
)
```

2.3.3. Die closed-serial-clause und ihr Wert

Eine serial-clause, eingeschlossen in *(* und *)* oder *begin* und *end*, nennen wir eine closed-serial-clause, sie heiszt aber offiziell: closed-clause.
Der Wert einer closed-serial-clause ist stets der Wert ihrer eingeschlossenen serial-clause, damit also der Wert ihres letzten unit.
Im zweiten Beispiel des vorigen Paragraphen ist der Wert der dort angegebenen closed-serial-clause (also das Ganze, inklusive *begin* und *end*) der Wert *void* (erste Fassung des Beispiels) oder der Wert, den *sd* liefert (zweite Fassung).

Wie aus dem Überblick von 2.1.3. hervorgeht, kann eine closed-serial-clause selbst als ein unit (z.B. in einer sie umfassenden closed-serial-clause) auftreten. Schon deshalb musz eine closed-serial-clause einen Wert besitzen.

Man beachte, dasz jedes unit (sei es ein 'open-unit' oder ein 'closed-unit') geklammert werden kann und dann eine closed-clause bildet (die das geklammerte unit als einzige phrase enthält).
Auf diese Weise werden in ALGOL68 die ALGOL60-Begriffe 'block', 'compound-statement' und 'geklammerter Ausdruck' zu einem Begriff closed-clause generalisiert.

Man vergleiche:

```
y := i × n

y := i × (n)

y := i × ( m+n )

y := i × (int m,n; read((m,n)); m+n)

y := i × row(k)

y := i × row( (int k:=1;
                for i from 2 to upb row
                do if row(i)>row(k) then k:=i fi od;
                k ) )
```

2.4. Deklaration und Identifizierung

2.4.1. Deklarationen (declarations)

In ALGOL68 hat der Programmierer die Möglichkeit, neben identifiers auch mode-indications und Operatoren sowie deren Prioritäten in sein Programm einzuführen. Diese Einführung geschieht mit Hilfe von declarations, die dem indicator eine bestimmte Bedeutung, seine "property", beigeben.

Es gibt vier Arten von declarations:

	declaration	"property"
a)	identifier-declaration	der identifier erhält einen Wert eines bestimmten "mode"
b)	mode-declaration	der mode-indication gibt einen bestimmten "mode" an
c)	operation-declaration	der Operator erhält eine "routine" mit einem oder zwei Parametern (monadischer oder dyadischer Operator)
d)	priority-declaration	der Operator erhält eine bestimmte Priorität

Ein label-identifier wird dagegen nicht in einer declaration, sondern durch den label, in dem er auftritt, definiert: seine "property" ist die Eigenschaft, als Sprungziel verwendet zu werden.

In den Kapiteln 3, 5 und 6 werden wir die Abarbeitung von declarations im einzelnen behandeln. Wir beschränken uns hier auf die Angabe einiger declarations, die bereits im Abschnitt 1.4 auftraten.

a) identifier-declaration:

real next, sum:=0, squm:=0

Hierdurch werden *next*, *sum* und *squm* als *real*-Variable definiert, sie erhalten also jeweils Werte vom "mode" *ref real*. Dabei werden den *ref reals* von *sum* und *squm* jeweils der Wert von *0* zugewiesen.

(1:n)real row

Hierdurch wird *row* als eine *()real*-Variable definiert, sie erhält also eine Adresse, die sich auf eine Reihe von n *real*-Werten bezieht. Der "mode" dieser Adresse ist also *ref ()real*.

employee director, employee

Hierdurch werden *director* und *employee* als *employee*-Variable definiert, sie erhalten also jeweils Werte vom "mode" *ref employee*. Man beachte, dasz hierbei *employee* ein "defining-identifier", während dagegen *employee* ein "applied-mode-indication" ist. Für den "defining-mode-indication" siehe b).

proc compute = (()real r) quadr: begin serial-clause *end*
(vgl. (7) in 1.4.6.)

Hierdurch erhält *compute* eine "routine" vom "mode" *proc(()real)quadr* .

b) mode-declaration

mode employee = struct (int entrance year ,
real salary , string name)

Hierdurch wird *employee* als ein indicator definiert, der den "mode" angibt, welcher durch den auf der rechten Seite angegebenen declarer spezifiziert wird.

c) operation-declaration

op implies = (bool a,b) bool: not (a and not b)

Hierdurch wird der indicator *implies* als ein Operator definiert, der eine "routine" vom "mode" *proc(bool,bool)bool* erhält.

d) priority-declaration

prio implies = 1

Hierdurch erhält der Operator *implies* die Priorität 1 .

2.4.2. Identifizierung

Wie wir in 2.3.3. gesehen haben, kann innerhalb einer serial-clause selbst wieder eine serial-clause (dann natürlich als closed-serial-clause) auftreten. In dieser inneren serial-clause können dann auch wieder declarations vorkommen.

Beispiel:

```
begin  real x ;   x := 3.14 ;
       begin  bool y ;   y := true ;
              print ( y )
       end ;
       print ( x )
end
```

Man beachte hierbei, wie die "applied-identifier" ihre "defining-identifier" identifizieren. Der Identifizierungsprozesz wird dabei nicht durch die Tatsache beeinfluszt, dasz in der inneren serial-clause *y* und in der äuszeren serial-clause *x* auftritt. Wir können innerhalb der inneren serial-clause sogar *y* durch *x* ersetzen, ohne dasz sich dadurch etwas ändert:

```
begin  real x ;   x := 3.14 ;
       begin  bool x ;   x := true ;
              print ( x )
       end ;
       print ( x )
end
```

Der identifier *x* in der äuszeren serial-clause und der identifier *x* in der inneren serial-clause haben jedoch verschiedene "properties": das eine *x* ist eine *real*-Variable, während das andere *x* eine *bool*-Variable ist.

In dem folgenden Beispiel beachte man, wie der mode-indication *rebo* sowie der Operator *implies* identifiziert werden:

```
begin  mode rebo  =  real ;
       op implies = (real a,b) bool: a>b ;
       rebo x , y ;
       x := 3.14 ;  y := 2.71 ;
       begin  mode rebo  =  bool ;
              op implies = (bool a,b) bool:
                              not ( a and not b ) ;
              rebo x , y ;
              x := true ;  y := false ;
              if x implies y then print(x)
                             else print(y)
              fi
       end ;
       if x implies y then print(x)
                      else print(y)
       fi
end
```

Durch die Identifizierung wird zu einem "applied-indicator" sein "defining-indicator" gesucht. Dieses Aufsuchen beginnt dabei stets in der kleinsten serial-clause, die den "applied-indicator" enthält. Führt die Suche in dieser serial-clause nicht zum Erfolg, so wird in der die bisher betrachtete serial-clause umfassenden serial-clause weiter gesucht.
Spätesten in der äuszersten serial-clause führt die Suche zum Erfolg, es sei denn, der indicator war standardmäszig definiert worden (vgl. auch Kapitel 7).

Durch die Identifizierung wird also entschieden, welche "property" ein "applied-indicator" erhält. Die Identifizierung von Operatoren ist im allgemeinen nicht so einfach, wie sie hier in dem Beispiel dargestellt wurde. Es müssen dabei nämlich noch die "modes" der Parameter mit in Betracht gezogen werden. (vgl. 5.1.4.).

2.5. Die conditional-clause

Mit Hilfe der conditional-clause hat der Programmierer die Möglichkeit, abhängig von einem *bool*-Wert, von zwei Programmteilen einen auszuwählen, in dem dann die "elaboration" seines Programms fortgesetzt wird.

2.5.1. Der äuszere Aufbau einer conditional-clause

Eine conditional-clause hat folgenden allgemeinen Aufbau:

```
if enquiry-clause then  in-choice-clause
                  else  out-choice-clause
fi
```

Hierbei kann die out-choice-clause auch fehlen, so dasz eine conditional-clause auch die Form haben kann:

```
if enquiry-clause then  in-choice-clause fi
```

Die fehlende out-choice-clause entspricht dann einem *skip* (vgl. 4.1.6.).

Eine andere, der obigen äquivalente, Schreibweise für eine conditional-clause ist:

```
( enquiry-clause | in-choice-clause | out-choice-clause )
```

bzw.

```
( enquiry-clause | in-choice-clause )
```

Beide Schreibweisen haben (abhängig vom Kontext) ihre Vorteile, und wir werden sie beide benutzen, mit einem gewissen Vorzug der ersten.

Die in-choice-clause und die out-choice-clause sind jeweils serial-clauses; in ihnen dürfen also declarations, units und labels auftreten (vgl. letztes Beispiel in 2.6.3). Die enquiry-clause ist ebenfalls eine serial-clause, jedoch dürfen in ihr keine labels auftreten, so dasz aus der in-choice-clause bzw. out-choice-clause nicht in die enquiry-clause zurückgesprungen werden kann.

Beispiel:

```
if int i; read(i); i≥0      then print (i); (i=0 | goto lab)
                            else print(-i)
fi
```

Dies ist eine gültige Form einer conditional-clause, wobei durch den jump *goto lab* an eine Programmstelle auszerhalb der gesamten conditional-clause gesprungen wird.
Nicht erlaubt ist dagegen:

```
if int i; lab:read(i); i≥0 then print (i); (i=0 | goto lab)
                           else print(-i)
fi
comment this is syntactical incorrect: the enquiry-clause
        contains a label comment
```

Die "elaboration" einer conditional-clause beginnt stets mit der "elaboration" ihrer enquiry-clause. Diese enquiry-clause musz dabei stets ein *bool* (also *true* oder *false*) liefern. Dies bedeutet also: das letzte unit der enquiry-clause musz ein *bool* liefern.
Wenn die enquiry-clause *true* liefert, so wird die in-choice-clause und nicht die out-choice-clause abgearbeitet; liefert dagegen die enquiry-clause *false*, so wird die out-choice-clause und nicht die in-choice-clause abgearbeitet.

2.5.2. Der Wert einer conditional-clause

Der Wert einer conditional-clause ist entweder der Wert ihrer in-choice-clause oder der Wert ihrer out-choice-clause, je nachdem, ob ihre enquiry-clause *true* oder *false* liefert.

Für die enquiry-clause gelten *if* und *then* bzw. (und |, für die in-choice-clause gelten *then* und *else* oder *fi* bzw. | und | oder) und für die out-choice-clause gelten *else* und *fi* bzw. | und) jeweils als Klammerungspaar.

Es kann jede conditional-clause, deren Wert nicht *void* ist, als Operand in einer formula auftreten.

Beispiel:

if $a<b$ *then* 3.14159 *else* 2.71828 *fi*
+
if $u \neq v$ *then* 0 *else* 1 *fi*

oder kürzer (was hier naheliegender erscheint):

$(a<b \mid 3.14159 \mid 2.71828) + (u \neq v \mid 0 \mid 1)$

Wenn die conditional-clause eine Adresse liefert, so darf sie auch als linke Seite einer assignation (also: als destination) vorkommen.

Beispiel:

if $n=0$ *then* x *else* y *fi* := 3.14159

Die beiden letzten Beispiele kann man kombinieren zu einem:

$(n=0 \mid x \mid y) := (a<b \mid 3.14159 \mid 2.71828) + (u \neq v \mid 0 \mid 1)$

Man beachte hierbei die acht verschiedenen Zuweisungen, die möglich sind.

2.5.3. Schachtelung von conditional-clauses

Da die drei inneren Bestandteile einer conditional-clause jeweils serial-clauses sind, so dürfen sie insbesondere selbst wieder conditional-clauses sein.
Wir geben im folgenden einige Konstruktionen an, die möglich sein können.

Es seien im folgenden C1, C2, --- stets enquiry-clauses, die einen *bool* liefern. Desweiteren seien CiT bzw CiF jeweils in-choice-clauses bzw. out-choice-clause , die zu der enquiry-clause Ci gehören (i=1,2,3,---).

a) *if* *if* C1 *then* C1T *else* C1F *fi* *then* CT *else* CF *fi*

Diese Konstruktion wird nicht allzu oft vorkommen. Man beachte hierbei, dasz C1, C1T und C1F alle drei jeweils ein *bool* liefern müssen. Es ist leicht einzusehen, dasz nur dann CT abgearbeitet wird, wenn entweder C1 und C1T beide *true* oder C1 *false* und C1F *true* liefern. Dagegen wird CF abgearbeitet, wenn entweder C1 *true* und C1T *false* oder C1 und C1F beide *false* liefern.

Beispiel:

```
if if n=0 then a<b else u≠v fi then x:= 3.14159; y:= 0
                                else x:= 2.71828; y:= 1
fi
```

b) *if* C1 *then* *if* C2 *then* C2T *else* C2F *fi* *else* C1F *fi*

Auch diese Konstruktion wird uns nicht allzu oft beschäftigen. Was passiert ist folgendes:
Liefern C1 und C2 beide *true*, so wird C2T abgearbeitet, liefern C1 *true* und C2 *false*, so wird C2F abgearbeitet. Liefert C1 *false*, so wird stets C1F abgearbeitet.

Beispiel:

```
if n=0 then if a<b then x:= 3.14159 else x:= 2.71828 fi
                   else y:= 0
fi
```

Man beachte, dasz in beiden obenstehenden Konstruktionen a) und b) die *if*-*fi*-Konstruktion (jede conditional-clause hat dadurch eine geschlossene Form) jede Mehrdeutigkeit ausschlieszt.

Sehr oft jedoch benutzt man:

```
c)  if C1 then C1T
          else if C2 then C2T
                    else if C3 then C3T
                              else if C4 then C4T
                                         else C4F
                              fi
                    fi
          fi
    fi
```

Diese Konstruktion ist leicht zu verstehen. Das einzige, was jeden stört, ist das *fi fi fi fi*.
So kann man auch kürzer schreiben:

```
    if C1 then C1T
  elif C2 then C2T
  elif C3 then C3T
  elif C4 then C4T
         else C4F
    fi
```

Oder auch kürzer:

```
  ( C1 | C1T
  |: C2 | C2T
  |: C3 | C3T
  |: C4 | C4T
        | C4F
  )
```

Hierbei 'schluckt' jeweils das *elif* (Zusammenziehung von *else* und *if*) bzw. |: sein eigenes *fi* bzw.).
Diese Konstruktion ist der Sprache LISP entlehnt.

Beispiel:

```
    if n=0 then x:= 3.14159
  elif a<b then x:= 2.71828
  elif u≠v then y:= 0
         else y:= 1
    fi
```

Zum Schlusz ein Beispiel einer conditional-clause, in der alle drei inneren Bestandteile (enquiry-clause, in-choice-clause und out-choice-clause) jeweils serial-clauses darstellen, die aus mehr als einem unit bestehen.
Hier lohnt die Mühe, diesem Beispiel im einzelnen nachzugehen.

Beispiel:

Auf einem Eingabemedium sei eine nicht bekannte Anzahl positiver und negativer Zahlen ≠0 gegeben, denen eine 0 als Abschlusz folgt.
Von einem Programm soll folgendes ausgedruckt werden:

1) die Anzahl der positiven und negativen Zahlen,
2) die Anzahl der positiven Zahlen,
3) die Anzahl der negativen Zahlen,
4) Summe und Mittelwert der positiven Zahlen,
5) Summe und Mittelwert der negativen Zahlen,
6) Summe und Mittelwert der absoluten Werte aller Zahlen.

Wenn keine positiven Zahlen oder keine negativen Zahlen oder überhaupt keine Zahlen vorhanden sind, so soll das Programm auch diese Fälle entsprechend berücksichtigen.

```
begin int num:= 0, pos:= 0, neg:= 0;
     real absum:= 0, posum:= 0, negum:= 0, x:= 1;
     while x≠0
     do absum plusab if read(x); num plusab 1; x>0
                    then pos plusab 1; posum plusab x; +x
                    elif x<0
                    then neg plusab 1; negum plusab x; -x
                    else num minusab 1; 0
                      fi
     od ;
     print ( ( num , pos , neg , newline ,
               posum , (pos≠0|posum/pos| "undefined") , newline ,
               negum , (neg≠0|negum/neg| "undefined") , newline ,
               absum , (num≠0|absum/num| "undefined")
             )
           )
end
```

An die Idee, dasz eine serial-clause stets einen Wert (einschlieszlich *void*) liefert, musz man sich vielleicht erst gewöhnen. Hat man sich jedoch daran gewöhnt, so erkennt man bald, dasz es sich hierbei um ein vortreffliches Ausdrucksmittel handelt.
Man beachte im obigen Beispiel weiter (wie schon erwähnt), wie *if* mit *then*, *then* mit *else* und *else* mit *fi* die jeweiligen serial-clauses einschlieszen und wie darüberhinaus das letzte unit innerhalb dieser Klammerungs-Paare den Wert der jeweils eingeschlossenen serial-clause bestimmt:

$x>0$ bestimmt den Wert der enquiry-clause,
$+x$ bestimmt den Wert der in-choice-clause,
$-x$ oder 0 bestimmen den Wert der out-choice-clause.

2.6. Die case-clause

Wie wir im letzten Abschnitt 2.5 gesehen haben, hat der Programmierer durch die conditional-clause die Möglichkeit, bei der Abarbeitung seines Programms eine von zwei serial-clauses auszuwählen.
Will der Programmierer unter mehr als zwei Programmteilen einen auswählen, in welchem dann die Abarbeitung seines Programms fortgesetzt wird, so kann er entweder die case-clause oder die conformity-clause benutzen. Bei der case-clause hängt die Auswahl von einem *int* ab, während sie bei der conformity-clause von einem "mode" abhängt.
Wir wollen in diesem Abschnitt vorerst nur die case-clause eingehender betrachten, während wir im Zusammenhang mit den "united modes" in 4.3.4. auf die conformity-clause eingehen werden.

2.6.1. Der äuszere Aufbau einer case-clause

Eine häufig auftretende Konstruktion (eine Art Verteiler) ist die folgende:

```
if i=1 then unit für i=1
       elif i=2 then unit für i=2
                elif i=3 then unit für i=3
                         elif i=4 then unit für i=4
                                  else serial-clause
                                       für i<1
                                      oder i>4
fi
```

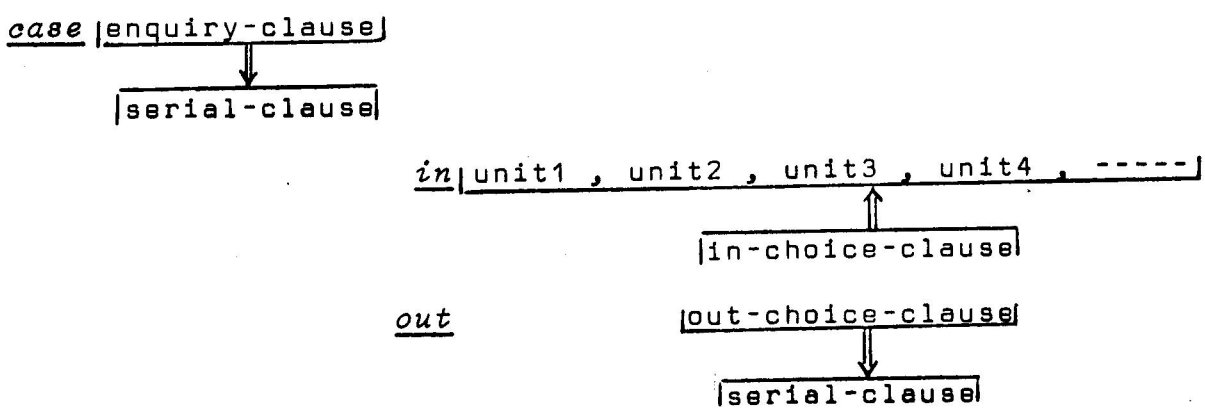
Dies ist natürlich für den Programmierer eine mühsame Schreiberei und, was viel schlimmer ist, für den Computer eine zeitaufwendige 'runtime'-Prüfung, insbesondere wenn *i* grosz ist und es somit viele Fälle gibt.
Semantisch äquivalent und für den Compiler leichter zu optimieren ist die case-clause:

```
case i in unit für i=1 , unit für i=2 ,
          unit für i=3 , unit für i=4
     out serial-clause für i<1 oder i>4
esac
```

Die case-clause und die conformity-clause (vgl. 4.3.4.) stellen Konstruktionen in ALGOL68 dar, wo das Komma "," nichts mit der Kollateralität zu tun hat: es wird stets genau eine der Möglichkeiten abgearbeitet, je nachdem, (im Falle einer case-clause) welchen *int*-Wert die enquiry-clause liefert (im obigen Beispiel also *i*) bzw. (im Falle einer conformity-clause) welchen "mode" das 'union' der enquiry-clause hat.

Die enquiry-clause (im obigen Beispiel besteht sie nur aus dem identifier *i*) und die out-choice-clause sind jeweils serial-clauses (wie bei einer conditional-clause). Hierbei musz dann die enquiry-clause stets entweder einen *int* (case-clause) oder ein *union* (conformity-clause) liefern.
Wie in einer conditional-clause kann die out-choice-clause fehlen, oder aber selbst eine neue case-clause (bzw. conformity-clause) sein. In diesem zweiten Fall (der bei einer conditional-clause dem Fall c in 2.5.3 entspricht) kann man *out* *case* bzw. |(ersetzen durch *ouse* bzw. |:, wobei das *ouse* bzw. |: sein *esac* bzw.) 'schluckt.

Die allgemeine Form einer case-clause (bzw. conformity-clause) ist also:



Für die Ineinanderschachtelung gilt allgemein:

```
case enquiry-clause1  in  in-choice-clause1
ouse enquiry-clause2  in  in-choice-clause2
ouse enquiry-clause3  in  in-choice-clause3
                     out  out-choice-clause3
esac
```

Oder kürzer:

```
( enquiry-clause | in-choice-clause | out-choice-clause )
```

beziehungsweise:

```
( enquiry-clause1 |  in-choice-clause1
|: enquiry-clause2 |  in-choice-clause2
|: enquiry-clause3 |  in-choice-clause3
                   | out-choice-clause3
)
```

2.6.2 Der Wert einer case-clause

Der Wert einer case-clause ist entweder der Wert des ausgewählten unit der in-choice-clause oder der Wert der out-choice-clause, jeweils abhängig von dem *int*, den die enquiry-clause liefert.

Beispiel:

```
real nfac := case int n; read (n); n+1
                 in 1 , 1 , 2 , 6 , 24 , 120 , 720 ,
                    5040 , 40320 , 362880 , 3628800 ,
                    39916800 , 479001600 ,
                    6227020800 , 24908083200
              ouse n-14
                 in 24908083200 × 15 ,
                    24908083200 × 15 × 16 ,
                    24908083200 × 15 × 16 × 17
                 out real pi:= 3.141592653589 ,
                          e := 2.718281828904 ;
                     print ( if n<0 then "undefined: n negativ"
                                    else
                             stirling:
                             sqrt ( 2×pi×n) × ( n/e)↑n
                             fi
                           )
              esac
```

Hierbei sind die units der in-choice-clause1 von einfacher Gestalt, nämlich Konstanten (integral-denotations), die units der in-choice-clause2 sind formulas. Solche units können aber im allgemeinen komplette closed-serial-clauses sein, in welchen declarations, weitere units und labels auftreten dürfen.
Die enquiry-clause1 ist eine serial-clause, die enquiry-clause2 ist eine einfache formula; die out-choice-clause2 ist wieder eine serial-clause.

2.7. Die loop-clause

Mit Hilfe der loop-clause kann der Programmierer bestimmte Programmteile zyklisch wiederholen.

2.7.1. Der äuszere Aufbau einer loop-clause

Eine loop-clause hat die folgende allgemeine Form:

```
for control-identifier from start-unit
                        by  step-unit
                        to  stop-unit
                     while enquiry-clause
  do serial-clause od
```

Hierbei gilt folgendes:

control-identifier	ist eine ganzzahlige Konstante, genauer: ein identifier, der stets ein *int* erhält;
start-unit step-unit stop-unit	sind jeweils units, die ebenfalls stets ein *int* liefern;
enquiry-clause	ist (wie bei der conditional-clause) eine serial-clause, die stets ein *bool* liefert;
serial-clause	die serial-clause zwischen *do* und *od* liefert stets ein *void* .

Man beachte hierbei, dasz in der enquiry-clause (nach *while*) keine labels auftreten dürfen, so dasz aus der zu wiederholenden serial-clause zwischen *do* und *od* nicht in die enquiry-clause zurückgesprungen werden kann.

Beispiel:

Auf einem Eingabemedium steht eine alternierende Folge von Zahlen, deren absolute Werte abnehmen. Man möchte die Summe derjenigen Zahlen bilden, deren absolute Werte noch gröszer oder gleich einem vorgegebenen *real*-Wert *eps* sind.

```
real sum:= 0; int n; read (n); (1:n)real row; read (row);

for i from 1 by 1 to n
while real term:= row(i); abs term ≥ eps
  do sum:= sum+term od
;
```

Durch das Paar *do*-*od* wird die serial-clause in jedem Fall geklammert, auch wenn sie nur (wie etwa im obigen Beispiel) aus einem einzigen unit besteht. Gerade wenn die zu wiederholende serial-clause aus mehreren ineinander geschachtelten closed-units (vgl. 2.1.2) besteht, wird durch diese *do*-*od*-Klammerung sowohl dem Compiler als auch dem Programmierer die Arbeit dann erleichtert, wenn sie feststellen wollen, wie weit der zu wiederholende Programmteil reicht.

Start-unit, step-unit und stop-unit werden stets nur einmal und zwar zu Beginn der "elaboration" der loop-clause kollateral berechnet, so dasz eine Veränderung ihrer Werte (z.B. durch eine Zuweisung in der enquiry-clause oder in der serial-clause) keinen Einflusz auf den Laufmechanismus der loop-clause hat.

Control-identifier erhält zu Beginn den Wert vom start-unit und wird nach jeder Abarbeitung der serial-clause um den Wert des step-unit solange erhöht, wie control-identifier $\leq$ stop-unit bei positivem step-unit bzw. control-identifier $\geq$ stop-unit bei negativem step-unit gilt (wenn nicht vorher schon die enquiry-clause den Wert *false* geliefert hat).

Gilt also control-identifier > stop-unit bei positivem step-unit bzw. control-identifier < stop-unit bei negativem step-unit oder aber liefert die enquiry-clause, die bei jedem Durchgang auf's neue abgearbeitet wird, einmal den Wert *false*, so wird die serial-clause nicht noch einmal abgearbeitet und die "elaboration" der loop-clause selbst ist damit beendet. Übrigens, wenn die enquiry-clause von Anfang an *false* liefert, so wird die serial-clause überhaupt nicht abgearbeitet.

Eine zwischen *do* und *od* stehende serial-clause liefert immer *void*, auch wenn sie in einer loop-clause wiederholt abgearbeitet wird. Natürlich liefert dann die loop-clause selbst auch immer *void*.

2.7.2. Verschiedene Formen einer loop-clause

In gewissen Fällen kann man einige Teile der loop-clause weglassen. Dies sind im einzelnen die folgenden:

a) Wenn der control-identifier weder in der enquiry-clause noch in der serial-clause zwischen *do* und *od* auftritt, so kann man

for control-identifier

weglassen.

b) *from 1* und *by 1* können weggelassen werden.

c) Wenn der control-identifier nicht auf seine obere (bzw. untere) Grenze stop-unit abgefragt wird, kann man *to* stop-unit weglassen.

d) *while true* kann weggelassen werden.

Da man jedoch dem Compiler wenigstens mitteilen musz, welches Programmstück zyklisch wiederholt werden soll, so darf man *do* serial-clause *od* nicht weglassen. Eine loop-clause hat also mindestens die Form:

do serial-clause *od*

In diesem letzten Fall kann der Programmierer die "elaboration" der loop-clause nur noch durch einen Sprung aus ihr heraus beenden.

Auszer dieser Minimal-Form kann also eine loop-clause z.B. die folgenden Formen haben:

while enquiry-clause *do* serial-clause *od*
to stop-unit *do* serial-clause *od*
by step-unit *do* serial-clause *od*
from start-unit *do* serial-clause *od*
for control-identifier *do* serial-clause *od*

etc.

Das Beispiel vom vorigen Paragraphen 2.7.1. kann man also kürzer schreiben:

```
real sum:= 0; int n; read (n); (1:n)real row; read (row);
for i to n while real term:= row(i); abs term ≥ eps
do sum:= sum+term od ;
```

2.8. Die closed-collateral-clause

Wie wir in 1.3.5 gesehen haben, hat der Programmierer mit Hilfe der closed-collateral-clause die Möglichkeit, mehrere units völlig unabhängig voneinander abarbeiten zu lassen. Gerade dadurch kann der Programmierer unter Umständen dem Compiler die Gelegenheit geben, eine Code-Optimierung vorzunehmen sowie mehrere Prozessoren einzusetzen.

Übrigens, eine closed-collateral-clause heiszt offiziell im Report: collateral-clause.

2.8.1. Der äuszere Aufbau einer closed-collateral-clause

Eine closed-collateral-clause besteht aus mindestens zwei units, die jeweils durch ein and-also-token "," voneinander getrennt sind. Sie ist stets entweder durch *(* und *)* oder ***begin*** und ***end*** abgeschlossen.

Beispiel:

(u:= n + 1 , v:= n + 2 , w:= n + 3)

Die drei assignations werden also völlig unabhängig voneinander abgearbeitet und man kann sich dabei vorstellen, dasz tatsächlich drei verschiedene Prozessoren sich damit beschäftigen.
Es sei noch einmal daran erinnert, dasz die units keine Nebeneffekte (!) aufeinander ausüben sollten, da diese nicht zu kontrollieren sind (vgl 1.3.5). Sogar dann, wenn keine Code-Optimierung vorgenommen wird und es nur einen Prozessor gibt, bedeutet "collateral elaboration": Nebeneffekte sind hier "undefined", also in der Programmierpraxis: Nebeneffekte sind hier nicht erlaubt (!).

Die allgemeine Form einer closed-collateral-clause ist:

(unit1 , unit2 , unit3 , unit4 , ---------)

beziehungsweise:

begin unit1 , unit2 , unit3 , unit4 , --------- ***end***

2.8.2. Der Wert einer closed-collateral-clause

Der Wert einer closed-collateral-clause ist das Gesamtgebilde der Werte ihrer units. Dieses Gesamtgebilde ist dabei stets entweder ein "multiple value" (vgl. 1.4.4) oder ein "structured value" (vgl.1.4.5). Welches von beiden, wird dabei durch den Kontext bestimmt. Im Fall eines "multiple value" heiszt die closed-collateral-clause: row-display, im Fall eines "structured value" heiszt sie structure-display.

Beispiele:

a) *compl z :=(1.2 , 3.4) ;*

b) *(1:2)real littlerow :=(1.2 , 3.4) ;*

c) *(1:3,1:3)real square := ((0.1 , 0.2 , 0.3) ,*
(0.4 , 0.5 , 0.6) ,
(0.7 , 0.8 , 0.9)) ;

d) *(1:2)compl complexrow :=: ((1.2 , 3.4) ,*
(5.6 , 7.8)) ;

e) *(1:4)real rowoffour :=(x↑(y+2)+a , x↑(y+2)-a ,*
x↑(y+2)×a , x↑(y+2)/a) ;

In b), c), d) und e) steht auf der rechte Seite ein row-display. In a) steht auf der rechte Seite eine structure-display. In c) sind die units des row-display selbst wieder row-displays, in d) dagegen sind die units, weil ein *compl* eine "structured value" ist, structure displays.
In e) hat wegen der Kollateralität ein 'schlauer' Compiler die Möglichkeit zu optimieren (nämlich *x↑(y+2)* zuerst und nur einmal zu berechnen).

2.9. Der completer

Wir wiesen schon daraufhin, dasz in den meisten Fällen der Wert einer serial-clause und damit auch der Wert einer closed-serial-clause der Wert seines letzten unit ist, es jedoch hiervon eine interessante Ausnahme gibt.
Eine Ausnahme ist es eigentlich nicht; dies hängt davon ab, was man unter 'letztes' unit verstehen will. Versteht man darunter das zuletzt bearbeitete (!) unit (innerhalb einer closed-serial-clause), so braucht dies nicht das in der textlichen Reihenfolge letzte unit zu sein: man kann mitten in einer closed-serial-clause aus ihr herausspringen.
Es kommt dann aber häufig vor, dasz man dabei den zuletzt berechneten Wert mitnehmen will. Der Wert eines jump ist aber immer *void*, ein jump kann also nichts anderes als *void* mitnehmen. Dazu haben wir dann in ALGOL68 den completer: *exit*.

Beispiel:

Vorgegeben seien ein *string* und ein *char*. Einer *bool*-Variable *b* wollen wir den Wert *true* zuweisen, wenn dieses *char* in dem *string* vorkommt. Darüberhinaus wollen wir den Index ausdrucken, der angibt, an welcher Stelle dieses *char* innerhalb des *string* vorkommt. Tritt das *char* dagegen in dem *string* überhaupt nicht auf, so wollen wir dem *b* den Wert *false* zuweisen.
Man kann dafür etwa das folgende Programm schreiben, welches einen completer verwendet:

```
string s; char c; read (s); read (c);
bool b:= ( int index;
           for i to upb s
           do  if c=s(i)
               then index:= i; goto present fi
           od;
   absent: false exit
   present: print (index);
           true
         ) ;
```

Hier passiert folgendes:
Wenn *c* in *s* nicht vorkommt (also *c=s(i)* stets *false* liefert), so gelangt man zu dem unit *false* und verläszt dann mit dem completer *exit* die closed-serial-clause. Sobald jedoch innerhalb der loop-clause festgestellt worden ist, dasz *c* in *s* auftritt, wird aus der loop-clause mit einem jump herausgesprungen zu dem label *present:* , wo dann der Index ausgedruckt wird.. Sodann wird die closed-serial-clause mit dem Wert *true* verlassen..

Wie aus dem Beispiel ersichtlich wird, kann man den completer als eine versteckte schlieszende Klammer in einer closed-serial-clause ansehen.

Wenn man diesen Sprung-Mechanismus verstanden hat, so wird man auch einsehen, dasz nach *exit* stets ein label stehen musz, da sonst das dem *exit* folgende unit nie (durch einen Sprung) erreicht werden kann.

2.10. Schluszbemerkung zum Satzbau

Es seien hier noch einmal die wichtigen Konzepte zusammengefaszt:

a) "serial" - "collateral" (vgl. 1.3.5)

b) declaration - unit (vgl. 2.1.3 , 2.2)

b1) eine declaration hat eine dienende Funktion:

b1a) sie gibt einem indicator (identifier oder indicant) eine lokale Bedeutung (sie definiert seine "property")

b1b) sie reserviert (in vielen Fällen) Speicherplatz für einen Wert eines bestimmten "mode"

b2) ein unit hat immer einen Wert, dieser kann sein:

b2a) *void*, dann können wir von einem statement sprechen (wie in ALGOL60)

b2b) *nonvoid*, dann können wir von einem expression sprechen (wie in ALGOL60)

c) closed-unit (vgl. 2.3.3 , 2.5 , 2.6 , 2.8)

c1) die closed-serial-clause ist selbst wieder ein unit, so erhalten wir die Blockstruktur in der Sprache; eine closed-serial-clause hat als unit auch immer einen Wert, auf diese Weise können grosze Programmstücke einen Wert liefern

c2) die closed-collateral-clause liefert ein Gebilde von Werten (nicht notwendig desselben "mode") ; durch die closed-collateral-clause kann dem Compiler Gelegenheit zur Code-Optimierung gegeben werden

c3) die conditional-clause und die case-clause liefern auch immer einen Wert und geben die Möglichkeit von Schachtelungen verschiedenster Art

c4) die loop-clause liefert stets *void*

d) die Sprache ist "orthogonal":
alle Kombinationen, die Bedeutung haben, sind erlaubt.

ALGOL 68

EINFÜHRUNG IN DIE ALGORITHMISCHE
SPRACHE ALGOL68

deel B

AEGU

REPRODUKTIE
FYSISCH LABORATORIUM

3. MODE UND IDENTIFIER

3.1 Die 'mode-maker' *ref* , ( ) , *struct* und *proc*

3.1.1 *nonref* , *refmod* und *amode*
3.1.2 Reihen ("multiple values")
3.1.3 Slices, indexers, trimmers, *at*, *lwb* und *upb*
3.1.4 Strukturen ("structured values")
3.1.5 Routinen
3.1.6 Definierbarkeit und gleichheit von "modes"
3.1.7 Slices und selections
3.1.8 Der äuszere Aufbau einer identity-declaration
3.1.9 Die "elaboration" einer identity-declaration

3.2 *nonref*-declarations

3.2.1 Deklaration von *nonref*-identifier
3.2.2 Deklaration von Prozeduren
3.2.3 Prozeduraufruf (ohne Parameter)
3.2.3 Prozeduraufruf ('call-by-value')

3.3 *refmod*-declarations

3.3.1 Deklaration von *refmod*-identifier
3.3.2 Prozeduraufruf ('call-by-reference')
3.3.3 Routinen die einen *refmod*-Wert liefern

3.4 "Generators"

3.4.1 *loc* und *heap*
3.4.2 Deklaration von Variablen
3.4.3 Initialisierung von Variablen
3.4.4 Deklaration von *ref* *procmode*-identifier
3.4.5 Rekursive Routinen

3.5 Weitere *refmod* Deklarationen

3.5.1 "Flexible-" und "transient-names"
3.5.2 Das Konzept eines "pointers"
3.5.3 Die identity-relation

3. MODE UND IDENTIFIER

Wir haben bereits bei einigen mode- und identifier-declarations gesehen, wie die 'mode-maker' verwendet werden (vgl. {5} --- {8} in 1.4 und 1.1.3).
In diesem Kapitel werden wir uns genauer ansehen, was die 'mode-maker'

ref , ( ) , *flex* , *struct* und *proc*

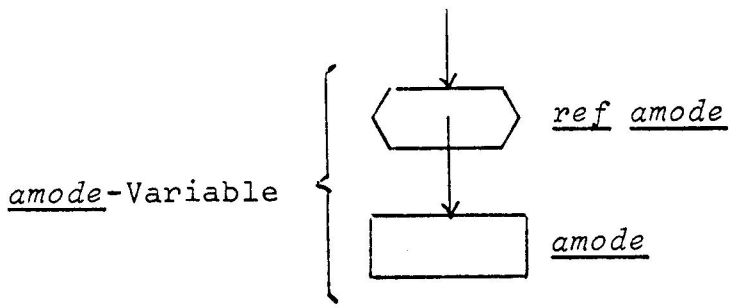
im einzelnen leisten (für *long*, *short* und *union* vergleiche Kap. 4)

Weiter werden wir uns eingehend mit der identifier-declaration beschäftigen die einem identifier einen Wert eines bestimmten "mode" zuordnet zu dem der identifier dann Zugriff ("access") hat. Wir werden dabei feststellen, dass die bisher betrachteten declarations für identifier (vgl. z.B. 1.4.8 {9}) hierunter fallen und dass auch die Parameterübergabe beim Aufruf einer Routine durch die Abarbeitung von identifier-declarations vorgenommen wird.

3.1. Die 'mode-maker' *ref* , ( ) , *struct* und *proc*

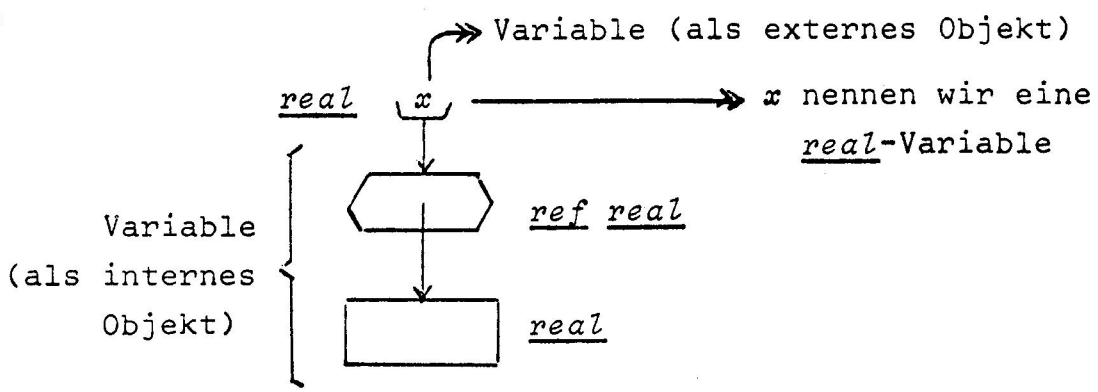
3.1.1. *nonref* , *refmod* und *amode*

Ein Paar, bestehend aus einer Adresse (*ref* *amode*) und dem Wert (*amode*), auf den sich diese Adresse bezieht, bezeichnen wir als eine Variable (vgl. 1.2.4 und 1.2.5):



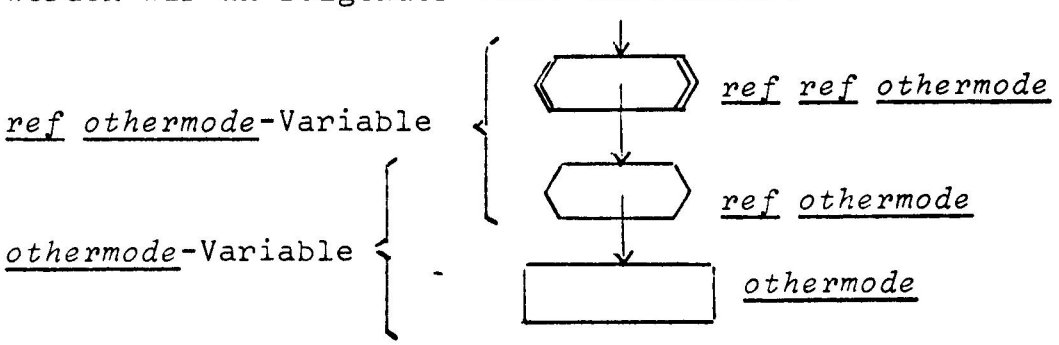
Auch einen identifier (also ein externes Objekt), der Zugriff hat zu der Adresse eines solchen Paares (deren "mode" also *ref* *amode* ist), nennen wir eine (*amode*-)Variable.

Beispiel:



Im Report wird solch eine declaration als variable-declaration bezeichnet.

Wegen der Orthogonalität kann man in ALGOL68 auch 'Ketten' von Referenzen (Adressierungs-Ketten) manipulieren: eine *amode* kann selbst wieder ein *ref* *othermode* sein, usw. Solche Referenzketten werden wir in folgender Weise darstellen:



Wir begegnen hier also Variablen auf verschiedenen Referenz-Stufen; Solche Adressierungs-Ketten werden wir in Kapitel 6 weiter betrachten (vgl. auch 3.5). Vorläufig beschränken wir uns auf Variable der niedrigsten Stufe.

Im folgenden bezeichnen wir mit '*nonref*' ('*nonref1*', '*nonref2*', usw.) irgendeinen Wert, der sich nicht mehr auf einen anderen Wert bezieht, wie z.B.

bool , *real* , ()*real* , (,)*real* , *compl* ,
man , *woman* , *fun* , *proc*(()*real*)*int* , usw.

Dagegen werden wir mit '*refmod*' ('*refmod1*', '*refmod2*', usw.) stets einen Wert bezeichnen, der sich auf einen anderen Wert bezieht, wie z.B.

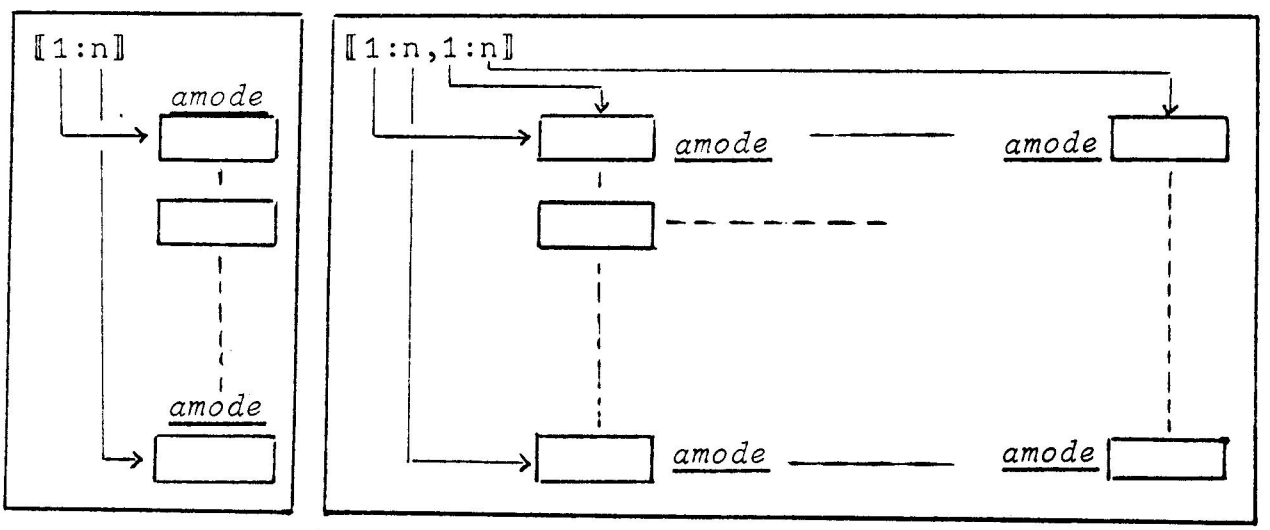
ref bool , *ref real* , *ref* ()*real* , *ref* (,)*real* , *ref compl* ,
ref man , *ref woman* , *ref fun* , *ref proc*(()*real*)*int* , usw.

Wollen wir jedoch einen Wert irgendeines beliebigen "mode" (also entweder ein '*nonref*' oder ein '*refmod*') bezeichnen, so schreiben wir '*amode*' ('*amode1*', '*amode2*', usw.).

Wir werden auch schreiben '*amode*-identifier' statt 'identifier der Zugriff hat zu einem *amode*-Wert', und '*ref amode*-identifier' statt 'identifier der Zugriff hat zu einem *ref amode*-Wert der sich bezieht auf einen *amode*-Wert'

3.1.2. Reihen ("multiple values")

Ein "multiple value" ist eine Reihe von Elementen desselben "mode" (vgl. 1.4.4). Die zum Auffinden der einzelnen Elemente im Speicher nötige Information wird durch den "descriptor" (auch "dope-vector") geliefert. Es enthält der "descriptor" die unteren und oberen Grenzen der (ein- oder mehr-dimensionalen) Reihe. Dabei ist der "descriptor" Bestandteil des "multiple value". Die folgende Bilder stellen "multiple values" dar:

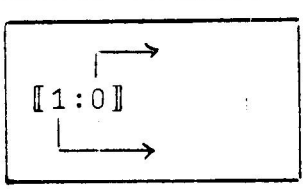


(1:n)amode (1:n,1:n)amode

Ein besondere "multiple value" ist der *string*:

mode string = *flex* (1:0)*char*

ein "multiple value" der 'atmen' kann. Der "descriptor" [1:0] (die untere Grenze ist gröszer als die obere Grenze) nennt man ein "flat descriptor": er beschreibt eine Reihe die (noch) keine Elemente enthält:



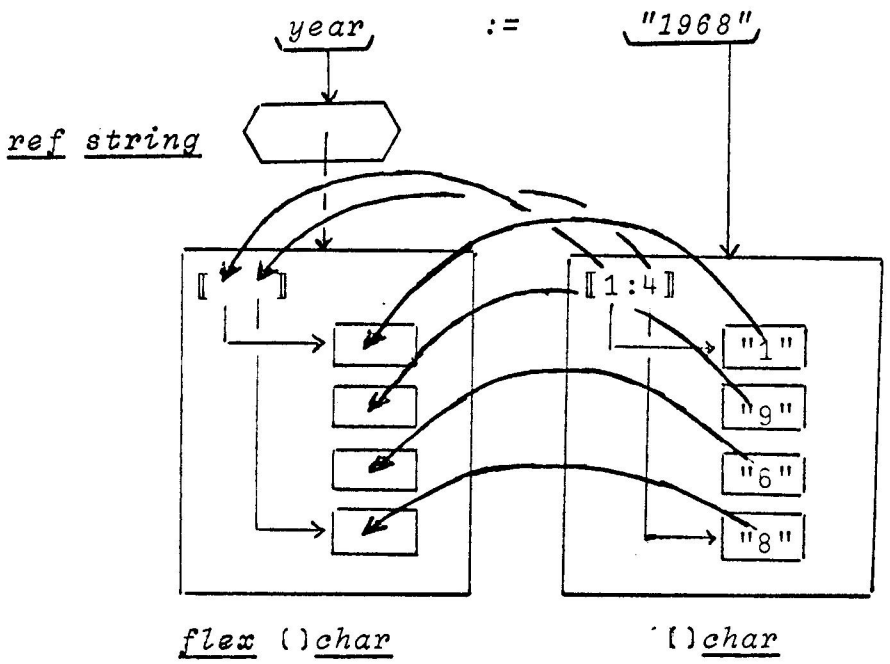
string

Weil die obere Grenze kleiner ist als die untere Grenze ("flat descriptor"), enthält ein *string* im Anfang noch keine Elemente (*char*s).

Eine *string*-Variable kann man jedoch eine ()*char* jeder Länge zuweisen:

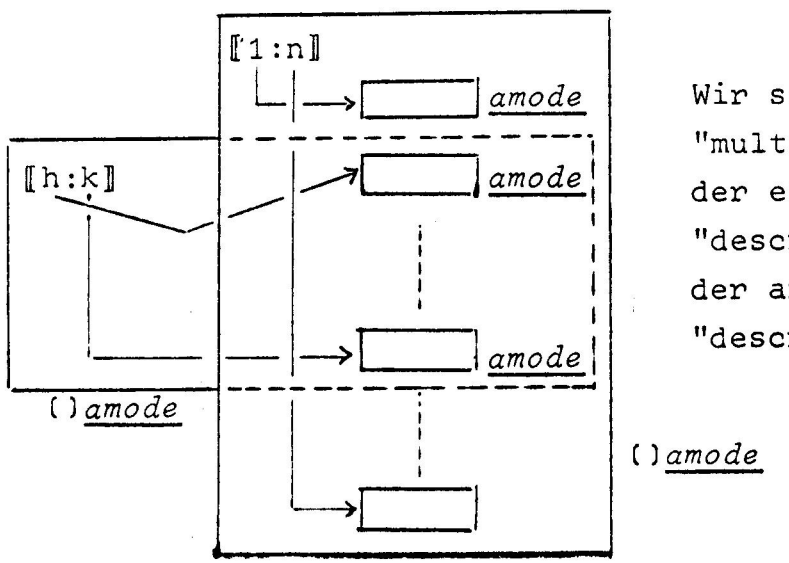
string year;

year := "1968"



Die Flexibilität (die Eigenschaft 'atmen' zu können) spielt erst dann eine Rolle, wenn man zuweist: das '*flex*' gehört also mehr wie ein Suffix zu der "name" als wie ein Prefix zu dem "multiple value" selbst.
Die "modes" *flex* []*char* und []*char* sind gleich, die "modes" *ref flex* []*char* und *ref* []*char* jedoch nicht: zu einem Wert vom "mode" *ref flex* []*char* kann man eine []*char* jeder Länge zuweisen (der "descriptor" rechts wird links kopiert, vgl. oben). Zu einem Wert vom "mode" *ref* []*char* werden die "descriptors" links und rechts verglichen und es folgt eine Fehlermeldung wenn sie nicht in allen Dimensionen gleich sind.

Wählt man einen Teil aus einem "multiple value" aus, so hat dieser ausgewählte Teil seinen eigenen "descriptor":

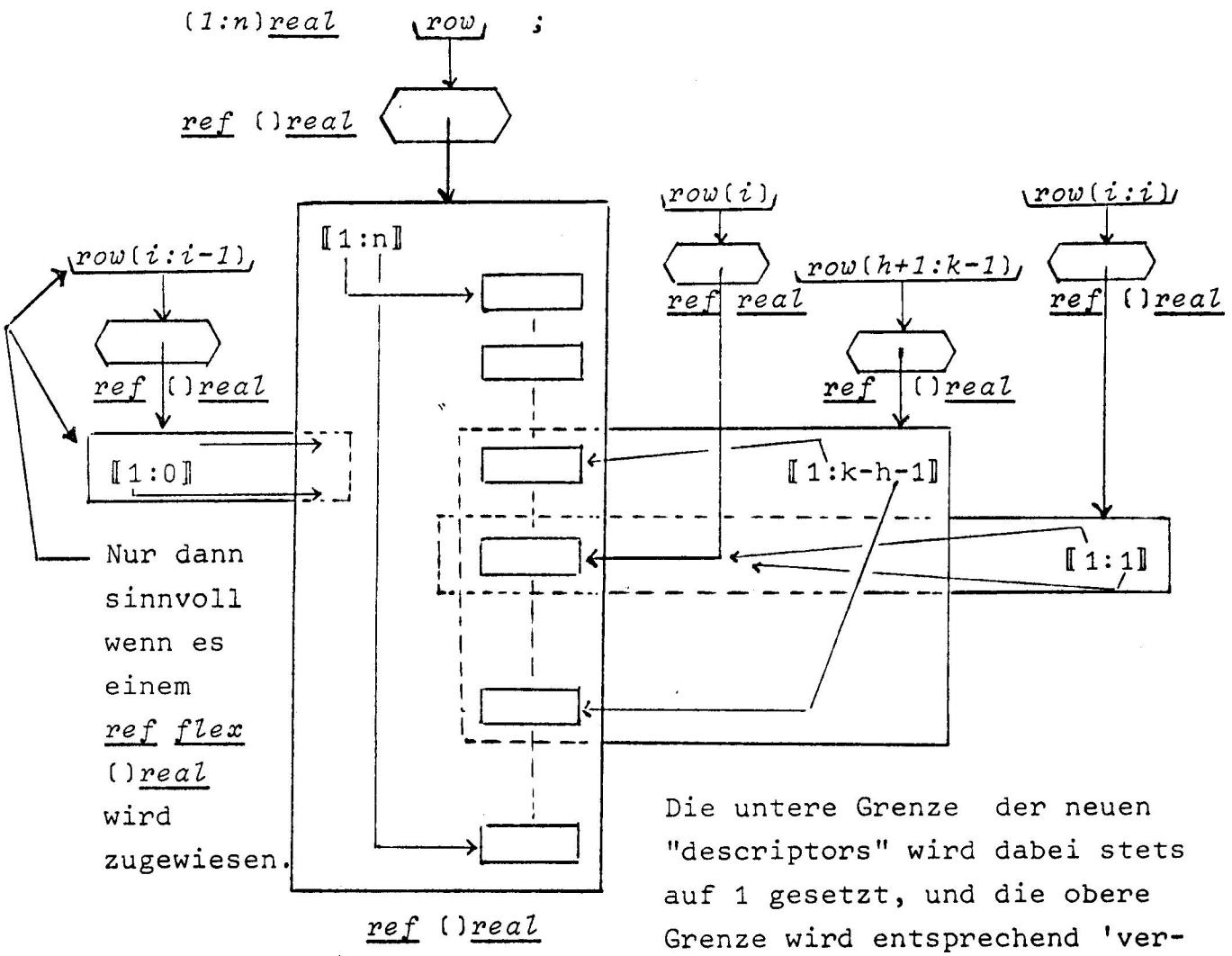


Wir sehen hier zwei "multiple values": der eine hat den "descriptor": [1:n], der andre hat den "descriptor": [h:k].

Hat ein identifier Zugriff zu der Adresse eines "multiple value", so erhält man durch "slicing" (vgl. 3.1.3) eine Adresse die sich bezieht auf ein Teil (bzw. ein Element) des "multiple value". Die allgemeine Regel (die auch gilt für "structured values", vgl. 3.1.4) in ALGOL68 ist:

> verfügt man über die Adresse eines Ganzen, so verfügt man über die Adressen der Teilen.

Beispiel (vgl. 1.1.4 {5}):



Die untere Grenze der neuen "descriptors" wird dabei stets auf 1 gesetzt, und die obere Grenze wird entsprechend 'verschoben' (vgl. 3.1.3).

Man beachte:

row	ist ein *ref* ()*real*-identifier
row(h+1:k-1)	ist ein *ref* ()*real*-unit
row(i:i)	ist ein *ref* ()*real*-unit ("descriptor" für 1 Element)
row(i:i-1)	ist ein *ref* ()*real*-unit ("flat descriptor")
row(i)	ist ein *ref real*-unit (kein "multiple value")

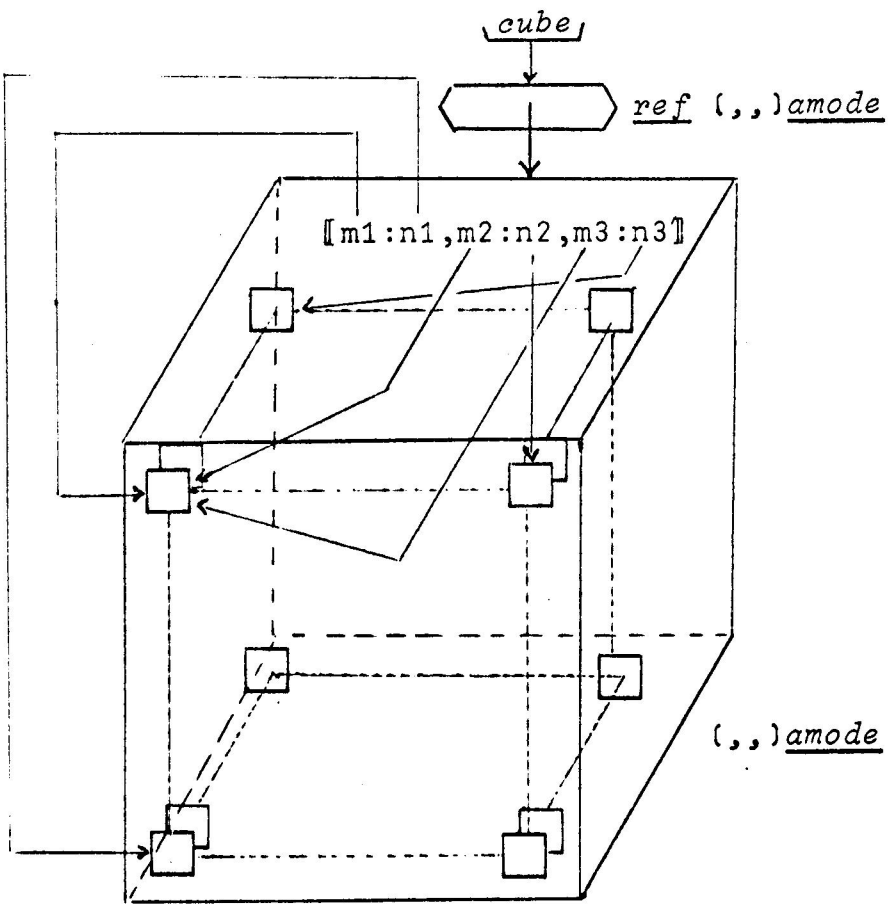
3.1.3 Slices, indexers, trimmers, *at*, *lwb* und *upb*

Konstruktionen wie *row(i)* und *row(h+1:k-1)* werden in ALGOL68 als slices bezeichnet. Mit Hilfe von slices hat der Programmierer die Möglichkeit, aus einer ein- oder mehrdimensionalen Reihe "subvalues", d.h. einzelne Elemente oder auch Teilreihen von gleicher oder kleinerer Dimension mit möglicherweise geänderten Grenzen auszuwählen.

Um verschiedene Arten von slices kennenzulernen, werden wir eine dreidimensionale Reihe betrachten, deklariert in folgender declaration:

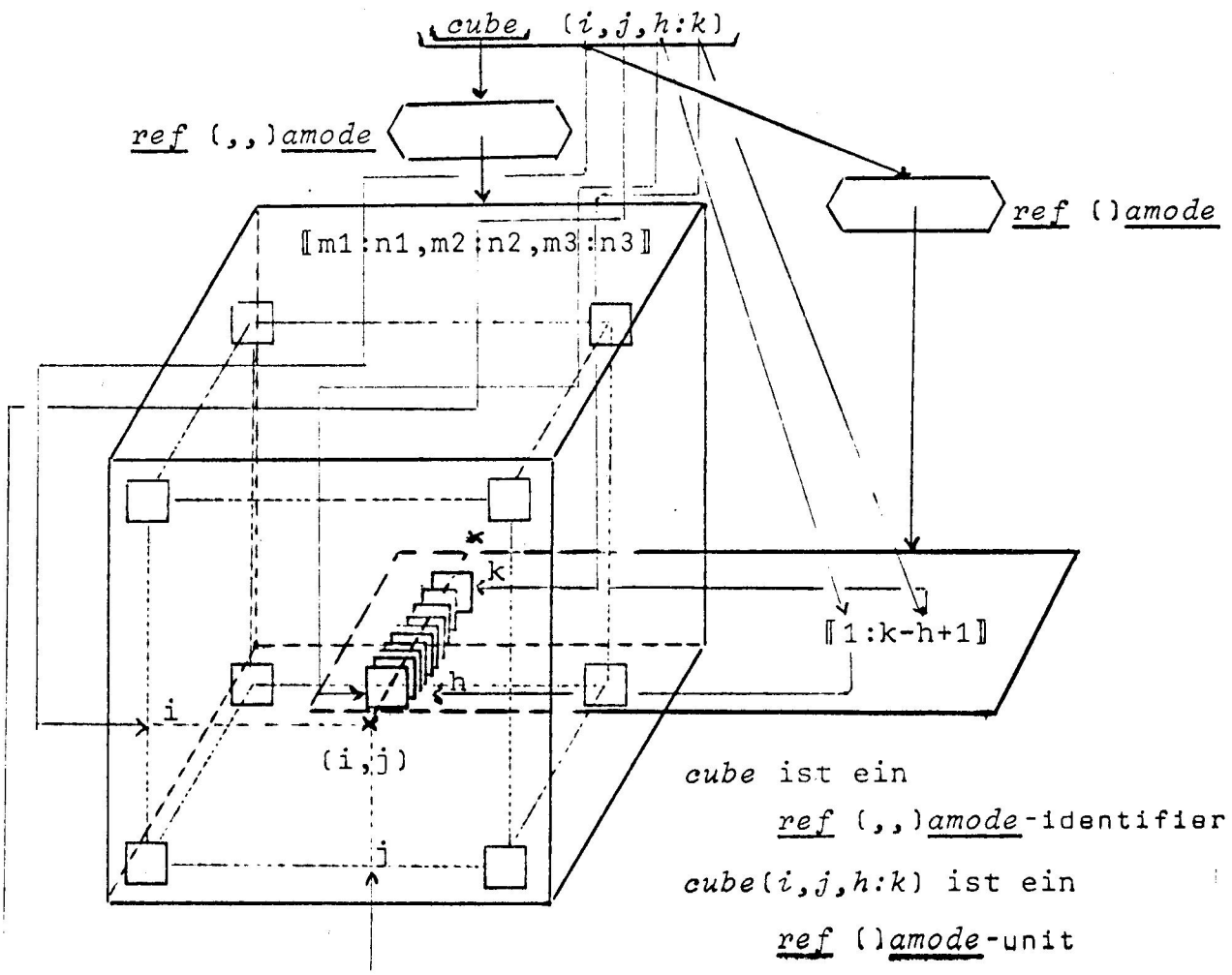
(m1:n1,m2:n2,m3:n3)amode cube ;

Die "elaboration" dieser identifier-declaration ergibt folgendes Bild:



Nach der "elaboration" seiner declaration erhält also *cube* die Adresse, die sich auf ein dreidimensionales "multiple value" vom "mode" (,,)*amode* bezieht.
Dieses "multiple value" besteht aus (n1-m1+1)x(n2-m2+1)x(n3-m3+1) Elementen vom "mode" *amode* und dem "descriptor" ⟦m1:n1,m2:n2,m3:n3⟧.

Will man nun einzelne Elemente oder Teilreihen aus diesem "multiple value" auswählen, so erreicht man dies durch Benutzung von indexers innerhalb eines slice.
Schreibt man z.B. das slice *cube*(*i*,*j*,*k*) und gilt dabei m1≤i≤n1, m2≤j≤n2 und m3≤k≤n3, so wird das Element (i,j,k) ausgewählt, genauer: *cube*(*i*,*j*,*k*) erhält die Adresse, die sich auf das Element (i,j,k) bezieht; dieses Element ist vom "mode" *amode*. Hierbei sind jeweils die indexers *i*, *j* und *k* "subscripts", die ein *int* liefern müssen.
Schreibt man dagegen das slice *cube*(*i*,*j*,*h*:*k*) und gilt dabei m1≤i≤n1, m2≤j≤n2 und m3≤h≤k≤n3, so wird hiermit eine eindimensionale Teilreihe ausgewählt und *cube*(*i*,*j*,*h*:*k*) erhält die Adresse, die sich auf diese Teilreihe des gesamten Gebildes bezieht.
Die "elaboration" von *cube*(*i*,*j*,*h*:*k*) liefert dann folgendes Bild:



Mit Hilfe des "descriptor" ⟦m1:n1,m2:n2,m3:n3⟧ des (,,)*amode*, wählt *cube*(*i*,*j*,*h*:*k*) die Elemente der angegebene Teilreihe aus. In diesem Beispiel wurden aus der i-ten Zeile (zweidimensionale Reihe mit n2-m2+1 Zeilen und n3-m3+1 Spalten) innerhalb der j-ten Spalte die Elemente von einschliesslich h bis einschliesslich k ausgewählt. Es wird dabei geprüft ob:

$$m1 \leq i \leq n1 \ , \ m2 \leq j \leq n2 \ , \ m3 \leq h \leq k \leq n3$$

Es wird dann für die ausgewählte Teilreihe einen neuen "descriptor" erzeugt und das slice *cube*(*i*,*j*,*h*:*k*) erhält die Adresse die sich auf diesen neuen "descriptor" (und durch diesen auf die Teilreihe) bezieht.

Man musz nun zwei wichtige Tatsachen wissen:

1) Es ist einen neuen "multiple value" entstanden. Diese neue ()*amode* und die ursprungliche (,,)*amode* haben zwar die ausgewählten k-h+1 Elementen mit einander gemein, sie sind jedoch verschieden weil sie verschiedene "descriptors" haben. Jedes slice erzeugt also einen neuen "multiple value", obwohl keine neue Elemente erzeugt werden (!): die *amode*-Elementen bleiben wer und wo sie sind, ein neuer "descriptor" wird geboren.

2) Die untere Grenze des neuen "descriptor" wird, wenn nicht anders angegeben (vgl. unten), auf 1 gesetzt und die obere Grenze dementsprechend auf k-h+1. Der durch *cube*(*i*,*j*,*h*:*k*) erzeugte "descriptor" ist also:

⟦ 1 : k-h+1 ⟧

Der indexer *h*:*k* innerhalb von *cube*(*i*,*j*,*h*:*k*) wird auch als trimmer bezeichnet.

Dabei kann ein trimmer aus einer unteren Grenze (im Beispiel *h*) und einer oberen Grenze (im Beispiel *k*) bestehen, die durch das up-to-symbol : voneinander getrennt sind.

Fehlen innerhalb eines trimmer entweder die untere oder die obere Grenze oder fehlen beide (es besteht also der trimmer aus : allein), so werden zur Auswahl der Elemente die entsprechende untere oder obere Grenze oder beide Grenzen von dem gegebenen "multiple value" übernommen.

In allen Fällen werden die ausgewählten Elemente von 1 an durchnummeriert. In den Beispielen *cube(i,j,h:)* , *cube(i,j, :k)* und *cube(i,j,:)* hat demnach der zugehörige "descriptor" jeweils die Form:

⟦1:n3-h+1⟧ , ⟦1:k-m3+1⟧ , ⟦1:m3-n3+1⟧

Neben *cube(i,j,:)* kann man auch *cube(i,j,)* (also ohne up-to-symbol :) schreiben. Auch hier werden zur Auswahl der Elemente die entsprechende untere und obere Grenze des gegebenen "multiple value" übernommen, jedoch werden die ausgewählte Elemente nicht von 1 an durchnummeriert. Der zu *cube(i,j,)* gehörige "descriptor" hat die Form:

⟦m3:n3⟧

Will andererseits der Programmierer die untere und damit die obere Grenze der ausgewählten Teilreihe selbst bestimmen (d.h. die untere Grenze soll nicht implizit auf 1 gesetzt werden), so kann er dies durch Angabe einer neuen untere Grenze am Ende des trimmer explizit angeben; ihr voran musz man das at-symbol <u>at</u> (oder auch @) schreiben. So hat der zu *cube(i,j,h:k* <u>*at*</u> *h)* zugehörige "descriptor" die Form:

⟦h:k⟧

In vielen Fällen ist dies gerade dasjenige das man vielleicht gern haben möchte (und möglicherweise auch erwartet hat); natürlich gibt es auch andere Möglichkeiten wie zum Beispiel:

cube(i,j,h:k <u>*at*</u> *m3)*	"descriptor": ⟦m3:k-h+m3⟧
cube(i,j,h:k <u>*at*</u> *0)*	"descriptor": ⟦0 :k-h⟧

Werden durch ein slice zwei- oder höherdimensionale Teilreihen ausgewählt, z.B. durch *cube(i,p:q,h:k)* , *cube(i,* <u>*at*</u> *1* ,) , *cube(i,,)* oder *cube(,,)* , so werden jeweils die einzelnen Grenzen wie oben angegeben berechnet und gesetzt.
Interessant vielleicht ist noch das slice *cube(*<u>*at*</u> *1,* <u>*at*</u> *1,* <u>*at*</u> *1)*; das ursprungliche "multiple value" (mit "descriptor" ⟦m1:n1,m2:n2,m3:n3⟧) wird in dieses slice beschrieben durch den "descriptor:

⟦1:n1-m1+1,1:n2-m2+1,n3-m3+1⟧

Dies kann man auch erreichen durch *cube(: , : , :)*.

Es wird all' denjenigen, die jetzt Kopfschmerzen haben, empfolen die neue untere Grenze stets durch *at* zu bestimmen (damit man wenigstens sicher ist).

Zusammenfassend können wir folgendes sagen:

cube(i,j,h:k)	ist gleichbedeutend mit	*cube(i,j,h:k at 1)*
cube(i,j,h:)	" " "	*cube(i,j,h:n3 at 1)*
cube(i,j, :k)	" " "	*cube(i,j,m3:k at 1)*
cube(i,j, :)	" " "	*cube(i,j,m3:n3 at 1)*
cube(i,j,)	" " "	*cube(i,j,m3:n3 at m3)*

Beispiel: (vgl. auch 3.3.1)

Es sei deklariert: *(0:m,0:n)real mat;*

Für eine Berechnung will man die i-te Reihe und j-te Spalte von *row* Element nach Element ändern, dabei braucht man jedoch die ursprungliche Werte aller Elementen der Reihe. Man kann dann zwei Reihen deklarieren:

(0:n)real rowi := mat(i,), (0:m)real colj := mat(,j) ;

Durch Zuweisungen an *rowi(j)* bzw. *colj(i)* ändert man die kopierte i-te Reihe bzw. j-te Spalte (ohne jedoch die ursprungliche Elemente zu ändern).

Nach der Berechnung kann man dann die neue Zeile und Spalte an *mat* zuweisen:

mat(i,):=rowi ; mat(,j):=colj ;

In vielen Fällen kennt man die aktuellen Werte der Grenzen eines "multiple value" nicht, zum Beispiel in einem routine-text, wenn die Reihe als formal-parameter auftritt. In 1.4.6 {7} haben wir schon gesehen dasz es hierfür die beiden Operatoren *lwb* und *upb* gibt; sie können sowohl monadisch wie auch dyadisch verwendet werden (vgl. auch 1.3.4). So liefern:

1 lwb cube , *1 upb cube* , *2 lwb cube* , *2 upb cube* ,
3 lwb cube und *3 upb cube*

die untere bzw. obere Grenzen in der 1ste, 2te bzw. 3te Dimension von *cube*. Statt *1 lwb cube* bzw. *1 upb cube* darf man auch (monadisch) schreiben: *lwb cube* bzw. *upb cube*; dies ist besonders geeignet für 1-dimensionalen Reihen.

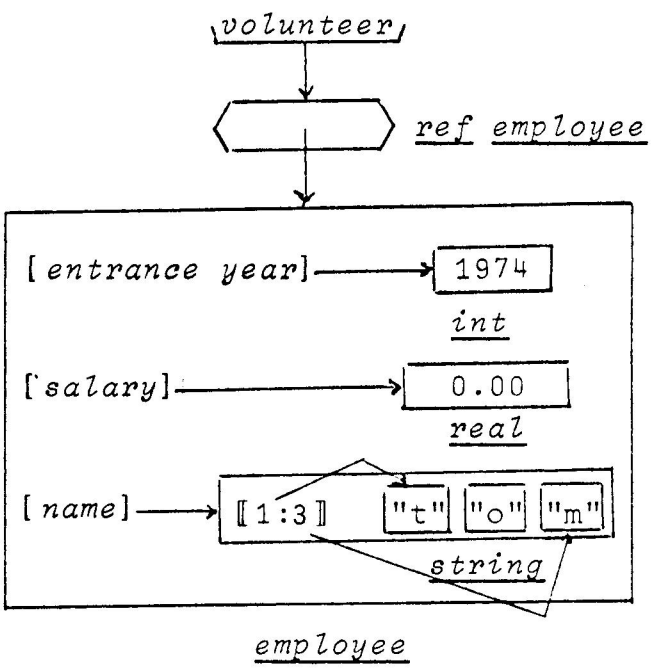
3.1.4. Strukturen ("structured values")

Ein "structured value" ist eine 'statische' Anordnung von Elementen, nicht notwendig desselben "modes". In ALGOL68 werden die einzelnen Elemente eines "structured value" als fields bezeichnet. Da die Anordnung 'statisch' ist (d.h. die Auswahl der Elemente ist dem Compiler schon in allen Einzelheiten bekannt, es gibt also keine "runtime"-Berechnung mehr), so braucht man zur Auswahl von fields keinen "descriptor": man wählt die Elemente (zur "compile-time") durch field-selectors.

Beispiel:

```
mode employee = struct ( int entrance year ,
                         real salary , string name ) ;
employee volunteer := (1974,0.00,"tom") ;
```

Wir erhalten folgendes Bild:



Wie man die einzelnen fields auswählt sieht man in folgende formulas, die alle den Wert *true* liefern:

```
entrance year of volunteer = 1974
salary of volunteer = 0.00
name of volunteer = "tom"
```

Die field-selectors sind keine identifiers, man kann sie also ohne Rücksicht auf schon deklarierten (und noch zu deklarieren) identifiers in einem *struct* verwenden. Innerhalb des *struct* müssen die field-selectors (um Mehrdeutigkeiten zu vermeiden) jedoch voneinander verschieden sein.

Ein *struct* kann als field in einem umfassenden *struct* auftreten.

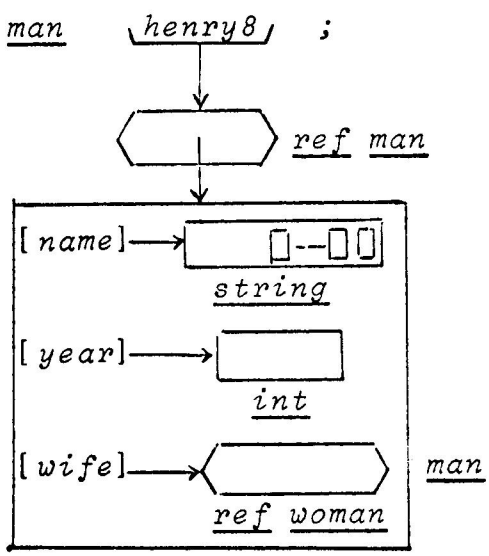
Beispiel:

```
mode record = struct ( int absence ,
                            working hours ,
                       real achievement ,
                     string rank ,
                   employee person ) ;

(1:n)record factory list ;

factory list (i) := ( 0 , 0 , 0.00 , "errandboy" , volunteer ) ;
```

Einen (höffentlich) glücklichen Tag für Tom:

```
record to be considered := factory list (i) ;

if absence of tobeconsidered = 0 and
   achievement of tobeconsidered > norm
then rank of tobeconsidered := "assistent";
     salary of person of tobeconsidered := 137.63
else rank of tobeconsidered := "vacant";
     name of person of tobeconsidered := ""
fi ;
```

Wesentlich komplizierter (und interessanter) wird es, wenn wir betrachten zum Beispiel:



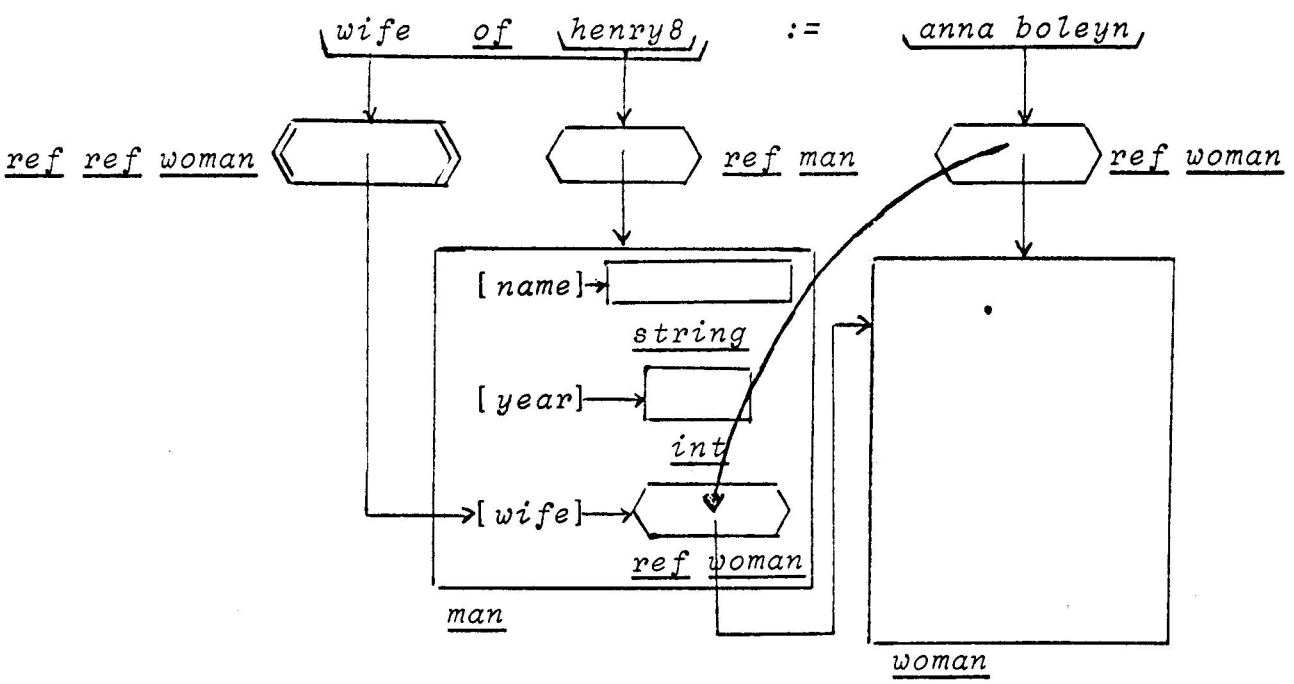
Manchmal musz man *henry8* eine *woman* zuweisen, zum Beispiel:

woman anna boleyn ;

Wir wissen schon (vgl. 3.1.2):

verfügt man über die Adresse eines Ganzen, so verfügt man über die Adresse der Teilen.

Nun ist das *wife*-field von *henry8* vom "mode" *ref woman*; die "selection" *wife of henry8* erhält also eine Adresse die sich bezieht auf eine *ref woman* und das bedeutet dasz der "mode" dieser Adresse *ref ref woman* sein musz. Wir gelangen somit auf natürliche Weise auf eine höhere *ref*-Stufe (vgl. Kap.6). Die assignation *wife of henry8 := anna boleyn* ergibt dann folgendes Bild:



Für Einzelheiten solcher Zuweisungen auf höheren *ref*-Stufen vergleiche 3.5.2, für Einzelheiten über die Verknüpfung von 'Listen' vergleiche Kapitel 6.

3.1.5. Routinen

Wie wir bereits in 1.3.3 gesehen haben, ist eine Routine ebenfalls ein Wert. Wie für die Werte von Grund-"modes", so haben wir auch für eine Routine eine Art von denotation. Diese heiszt in ALGOL68 routine-text: ein routine-text bestimmt stets eine Routine, ein Wert dessen "mode" anfängt mit '*proc*'. In dem Beispiel {7} von 1.4.6 haben wir bereits schon einen routine-text kennengelernt, der eine Routine liefert vom "mode"

proc (()*real*) *quadr*

Mehr im allgemeinen gibt es Routinen mit oder ohne Parameter, und eine Routine liefert nach ihrem Aufruf stets einen Wert (einschliesz-lich *void*). Eine Routine kann also einer der folgenden "modes" haben:

a)	*proc void*	kein Parameter, liefert *void*
b)	*proc amode*	kein Parameter, liefert *amode*
c)	*proc(amode1)void*	ein Parameter, liefert *void*
d)	*proc(amode1)amode*	ein Parameter, liefert *amode*
e)	*proc(amode1,amode2)void*	zwei Parameter, liefert *void*
f)	*proc(amode1,amode2)amode*	zwei Parameter, liefert *amode*
g)	*proc(amode1,amode2,amode3)void*	drei Parameter, liefert *void*
h)	*proc(amode1,amode2,amode3)amode*	drei Parameter, liefert *amode*
	und so weiter	und so weiter

Est ist also stets eine Routine ein *nonref*, auch dann wenn die Routine nach ihrem Aufruf ein *refmod* liefert. Der "mode" einer Routine werden wir mitunter durch *procmod* bezeichnen.

Allgemein kann ein routine-text das folgende Aussehen haben:

a) *void:* unit
b) *amode:* unit
c) *(amode1 formal1)void:* unit
d) *(amode1 formal1)amode:* unit
e) *(amode1 formal1, amode2 formal2)void:* unit
f) *(amode1 formal1, amode2 formal2)amode:* unit
g) *(amode1 formal1, amode2 formal2, amode3 formal3)void:* unit
h) *(amode1 formal1, amode2 formal2, amode3 formal3)amode:* unit
und so weiter

Das unit nach dem virtual-declarer *void* bzw. *amode* musz fähig sein einen *void* bzw. *amode* zu liefern (den "mode" gefordert von dem sogenannten virtual-declarer).

Die identifiers *formal1*, *formal2*, *formal3* usw. (vom "mode" *amode1*, *amode2*, *amode3* usw.) sind die formal-parameter der Routine: innerhalb des routine-text treten sie auf als Stellvertreter der (erst zum Aufruf bekannten) actual-parameter.

Die Routine, die die "elaboration" eines routine-text liefert, ist ein internes Objekt, ein 'Stück Objekt-Code'. Dieses interne Objekt (also ein Wert im Speicher des Computers, ein Wert in einem "locale") kann man sich durchaus als eine closed-clause vorstellen, die durch leichte Abänderung aus dem routine-text entsteht:

a) (*void*(unit))
b) (*amode*(unit))
c) ((*amode1* *formal1* =?1; *void*(unit)))
d) ((*amode1* *formal1* =?1; *amode*(unit)))
e) ((*amode1* *formal1* =?1, *amode2* *formal2* =?2 ; *void*(unit)))
f) ((*amode1* *formal1* =?1, *amode2* *formal2* =?2 ; *amode*(unit)))
und so weiter

Die besondere Klammern (und) dienen um anzudeuten dasz die Routine als internes Objekt noch keine closed-clause ist: die Fragezeichen ?1, ?2, usw. vertreten die (noch nicht bekannten) actual-parameter. Die Ausdrücke *void*(unit) und *amode*(unit) kennen wir schon als sogenannten casts (vgl. 2.2.2). Ein cast zwingt sein unit einen Wert des angegebenen "mode" zu liefern (vgl. 4.4.3). Die Routine liefert dann so einen Wert jedesmal wenn sie aufgerufen wird.

Beim Aufruf einer Routine werden die actual-parameter übergeben. Dabei entstehen dann folgende sogenannte identity-declarations:

c,d) *amode1* *formal1* = actual-parameter1 ;
e,f) *amode1* *formal1* = actual-parameter1 ,
amode2 *formal2* = actual-parameter2 ;
g,h) *amode1* *formal1* = actual-parameter1 ,
amode2 *formal2* = actual-parameter2 ,
amode3 *formal3* = actual-parameter3 ;

Eine identity-declaration ist eine der Möglichkeiten der identifier-declaration. In 3.1.8, 3.2 und 3.3 werden wir uns eingehend mit solchen declarations beschäftigen.

Zum Schlusz einige Beispiele:

In 1.1.3 deklarierten wir eine *fun* als ein *proc(real)real*. Ein routine-text der eine *fun* erhält ist:

```
(real x) real: sin(x)/x
```

↓

```
((real x = ? ; real(sin(x)/x)) )
```

Wird diese Routine aufgerufen mit z.B. dem actual-parameter *ax(u-v)*, so wird die closed-clause:

```
(real x = ax(u-v); real(sin(x)/x))
```

abgearbeitet (vgl. 3.2.4).

Ein routine-text der die Heirat zwischen *man* und *woman* schlieszt, ist:

```
(ref man he, ref woman she)void:
    begin wife of he := she ,
          husband of she := he
    end
```

↓

```
((ref man he = ?1, ref woman she = ?2;
      void (   wife of he := she ,
            husband of she:= he
          ) ) )
```

Die "elaboration" von identity-declarations wie

```
ref man he = henry8 , ref woman she = anna boleyn ;
```

wird in 3.3.2 behandelt.

3.1.6. Definierbarkeit und gleichheit von "modes"

Aus schon bekannten "modes" (auch wenn sie bereits aus anderen "modes" zusammengesetzt sind) kann man mit Hilfe der mode-declarations immer wieder neue "modes" definieren.
Dies ist jedoch nicht unbeschränkt möglich.
So gewährleistet die Syntax von ALGOL68, dasz wirklich nur solche "modes" definiert werden können, mit welchen der Computer (die ALGOL68-Maschine) auch etwas tun kann, d.h. "modes" die eine praktische Bedeutung haben. So sind mode-declarations die nur Fragen der Weltanschauung oder (der) Theologie formulieren, jedoch des weiteren ohne faszbare Bedeutung sind, von vornherein ausgeschlossen. Die Syntax wird zum Beispiel:

a) *mode allah = allah ;*

b) *mode what = that ,*
mode that = this ,
mode this = what ;

als Unsinn (d.h. als nicht-mode-declaration) ablehnen; der Compiler wird sie höffentlich als Fehler melden.

Auch höheren Blödsinn, wie zum Beispiel:

mode universe = (1:n)universe ,
mode neverready = struct(amode ok, neverready unselectable)

läszt der Compiler nicht zu.

Die Syntax zur Prüfung, ob eine mode-declaration sinnlos ist oder nicht, ist sehr kompliziert (und es ist nur für Spezialisten ein Vergnügen sie zu meistern). Für den Programmierer gibt es für diese Prüfung jedoch einen einfachen intuitiven Test:

läszt sich der Wert eines gewünschten "mode" als eine (zusammengesetzte) 'box', ohne perspektivische und andere zeichnerische Kunstgriffe, in endlicher Zeit in einem Bild darstellen, so wird auch der Compiler nichts beanstanden.

Überraschend ist vielleicht noch, dasz im Gegensatz zur mode-declaration

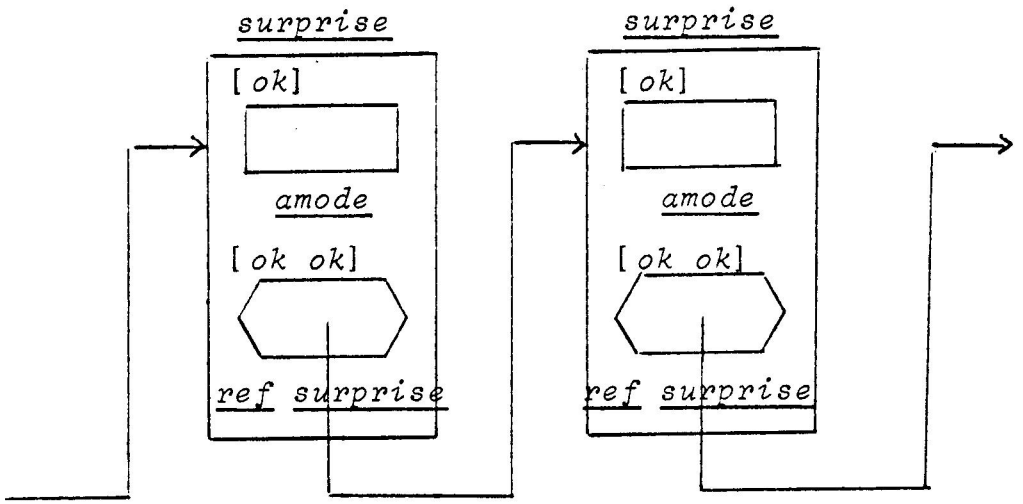
mode neverready = struct(amode ok, neverready unselectable)

(man versuche einmal hier eine entsprechende 'box' zu zeichnen),

die mode-declaration

mode surprise = struct(amode ok, ref surprise ok ok) ;

in jeder Hinsicht einwandfrei und z.B. für die Behandlung von Listen (vgl. Kap.6) sogar sehr nützlich ist:



Die *surprise* ist deswegen einwandfrei, weil *ref surprise* und *surprise* zwei verschiedenen "modes" sind: eine *surprise* ist ein *struct* mit zwei fields (eine 'box' worin zwei andere 'boxes'), ein *ref surprise* ist eine einfache Adresse.
Wir haben übrigens mit *man* und *woman* (auf indirekte Weise) bereits eine Surprise bekommen und hatten absolut keine Mühe die Bilder zu zeichnen.

Eine ähnliche Konstruktion ist

mode strange = struct(amode ok1, proc strange ok2)

Das Bild läszt sich auch hier ohne Tricks zeichnen. Was man dann zur "runtime" mit so einem *strange* anfangen will, ist eine ganz andere Frage.

In 1.2.3 haben wir schon darauf hingewiesen, dasz man stets entscheiden kann, ob zwei Werte vom gleichen "mode" sind oder nicht. Durch die Syntax von ALGOL68 wird überdies entschieden, ob durch zwei verschiedene mode-declarations der gleiche "mode" definiert worden ist. Trifft dies zu, so werden die entsprechenden mode-indications im weiteren Verlauf des Programms als gleichbedeutend behandelt.

Hat man zum Beispiel die mode-declarations

```
mode compl   = struct(real re, im) ;
mode complex = struct(real re, im) ;
mode xelpmoc = struct(real im, re) ;
```

so werden durch *compl* und *complex* der gleiche "mode" definiert (alsob man deklariert hätte: *mode complex = compl*), jedoch definiert *xelpmoc* einen anderen "mode", obwohl auch *xelpmoc* zwei *real*-fields und die beiden field-selectors *re* und *im* enthält.

Auch bei komplizierteren "modes" läszt sich der Compiler nichts vormachen, so werden von ihm z.B. nach

```
mode simple = struct(ref simple selector) ;
mode clumsy = struct(ref struct(ref simple selector)selector)
                          \____________________/
                                   simple
```

simple und *clumsy* als gleichbedeutend behandelt.

So kann man also, abgeschirmt durch die Syntax, alle diejenigen mode-declarations verwenden, die man für nötig hält.

Ist der Speicher grosz genug, so kann man z.B. schreiben:

```
int manymany; read(manymany);
mode humanity = (1:manymany)human;
```

Der Compiler (unwissend um wieviel *manymany* es sich hier handelt) entwickelt diesen "mode" zu:

```
(1:manymany)union(man,woman)
```

und diesen weiter zu:

```
(1:manymany)union(struct(string name, int year,
                         ref woman wife),
                  struct(string name, int year,
                         ref man husband,
                         ref flex (1:0)human child)
                  )
```

Solange es noch Material zum Ausbau von Speichern gibt, kann man folgendes versuchen:

```
int many; read(many);
mode history = struct((1:many)humanity struggling people,
                      short real food,
                      long  real garbage,
                      int year of selfdestruction) ;
```

und der Compiler entwickelt ohne Rücksicht:

```
struct((1:many)(1:manymany)union(struct(string name,
                                        int year,
                                        ref woman wife) ,
                                 struct(string name,
                                        int year,
                                        ref man husband,
                                        ref flex (1:0)
                                        union(man,woman)
                                        child)
                                 ) struggling people,
          short real food,
          long  real garbage,
          int year of selfdestruction
         ) ;
```

Die Orthogonalität des "mode"-Konzept in ALGOL68 kennt keine andere Beschränkungen als diejenige welche gegeben sind durch die logisch-pragmatische Konsistenz und (in der "elaboration") die Möglichkeiten des disponibelen Speichers. Innerhalb dieser Grenzen kann man realisieren was man braucht.

3.1.7. Slices und selections

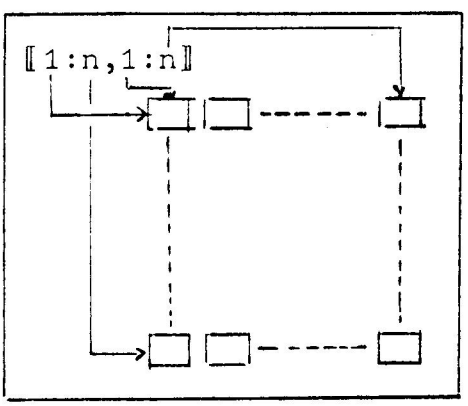
Manchmal sind Reihen, deren Elemente selbst wieder Reihen sind, für praktische Anwendungen von Interesse. Zum Beispiel:

mode *vecrow* = (*1:n*)*vector*

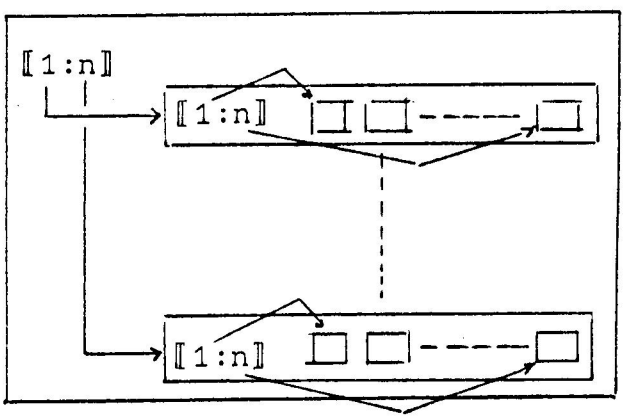
Der Compiler entwickelt also diesen "mode" zu:

(*1:n*)(*1:n*)*real*

Wir wollen nun *vecrow* vergleichen mit *matrix* :



matrix



vecrow

Deklariert man dann:

matrix *u* ;

vecrow *v* ;

so kann man ein einzelnes Element oder eine Teilreihe auf folgende Weise auswählen:

u(*i*,*j*) und nicht *u*(*i*)(*j*)

u(*i*,) und nicht *u*(*i*)

u(,*j*)

So man will, kann man hier statt *u*(*i*,*j*) auch schreiben:

u(*i*,)(*j*)

v(*i*)(*j*) und nicht *v*(*i*,*j*)

v(*i*) und nicht *v*(*i*,)

Auswahl einer 'j-ten' Spalte ist hier nicht möglich.

Betrachten wir nun die beide Fälle, wo die einzelnen "fields" einer Struktur selbst "multiple values" bzw. die Elemente eines "multiple value" selbst Strukturen sind.
In diesem Zusammenhang ergibt sich die Frage, ob der Compiler z.B. den Ausdruck:

selector <u>*of*</u> *struct* (*i*)

entweder als selection:

selector <u>*of*</u> (*struct*(*i*))

oder als slice:

(*selector* <u>*of*</u> *struct*)(*i*)

erkennt, ohne den "mode" von *struct* zu kennen ("mode-independent parsing").
Nach der Syntax von ALGOL68 ist

selector <u>*of*</u> *struct* (*i*)

stets eine selection, damit also gleichbedeutend mit

selector <u>*of*</u> (*struct*(*i*))

Betrachten wir nun die folgenden declarations:

<u>*mode*</u> <u>*compl*</u> = <u>*struct*</u>(<u>*real*</u> *re*,*im*) ,
<u>*mode*</u> <u>*corow*</u> = (*1*:*n*)<u>*struct*</u>(<u>*real*</u> *re*,*im*) , also (*1*:*n*)<u>*compl*</u>
<u>*mode*</u> <u>*costr*</u> = <u>*struct*</u>((*1*:*n*)<u>*real*</u> *re*,*im*) ;

<u>*corow*</u> *a*;

Da sich die Adresse von *a* auf eine Reihe bezieht, so kann man also das slice *a*(*i*) bilden.

Damit kann man dann auch die selection:
re <u>*of*</u> *a*(*i*) , also:
re <u>*of*</u> (*a*(*i*)) schreiben.

Überraschend ist vielleicht, dasz
re <u>*of*</u> *a*
syntaktisch korrekt ist. Hierdurch wird eine Reihe ausgewählt, deren Elemente gerade die n "fields" sind, die man mit Hilfe des field-

<u>*costr*</u> *b*;

Da sich die Adresse von *b* auf eine Struktur bezieht, so kann man also <u>nicht</u> *b*(*i*) schreiben.

Damit ist also *re* <u>*of*</u> *b*(*i*), was nur *re* <u>*of*</u> (*b*(*i*)) bedeuten könnte, syntaktisch inkorrekt.

Nicht überraschend ist jedoch dasz
re <u>*of*</u> *b*
syntaktisch korrekt ist. Es wird hierdurch das erste "field" (also eine Reihe von <u>*real*</u>s) eines <u>*costr*</u>

selector *re* jeweils aus den n Elementen von *corow* (dies sind also *compls*) ausgewählt.
Selbstverständlich hat der "descriptor" dieser so ausgewählte Reihe die Gestalt ⟦1:n⟧.

Demnach ist dann auch

(re of a)(i)

erlaubt.

Hierbei lohnt sich die Mühe festzustellen, dasz

(re of a)(i)
re of (a(i))
und
re of a(i)

jeweils den gleichen Wert liefern.

ausgewählt. Diese Reihe hat dann natürlich den "descriptor" ⟦1:n⟧.

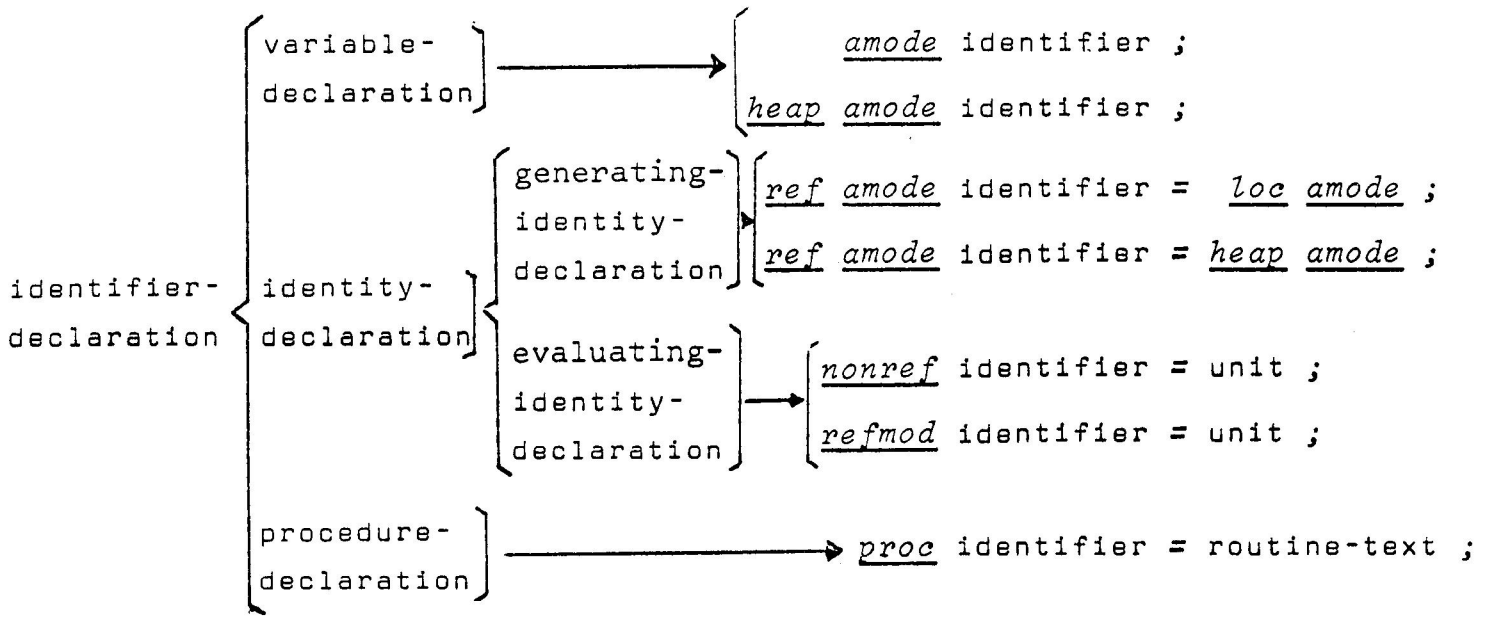
Demnach ist dann auch

(re of b)(i)

erlaubt.

3.1.8. Der äuszere Aufbau einer identifier-declaration

Eine identifier-declaration kann entweder eine variable-declaration, eine identity-declaration oder eine procedure-declaration sein. Im einzelnen kann eine identifier-declaration einer dieser drei Typen folgendes Aussehen haben:



Die Syntax erlaubt noch mehr: es können mehrere identifiers mit einem declarer (vgl. 3.1.9) vereinbart werden.
Zum Beispiel:

```
      real sum, squm, next;
 heap real hold1, hold2, hold3;
  ref real x = loc real , y = loc real , hold = heap real ;
      real pi = 3.14159 , e = 2.71828 ;
      proc sinx = (real x) real: sin(x)/x ,
           marry = (ref man he, ref woman she) void:
                   (  wife of he  := she ,
                   husband of she := he
                   ) ;
```

Wir unterscheiden (auf didaktischen Gründe) mehrere Fälle als syntaktisch nötig ist. So sind die generators *loc amode* und *heap amode* beide Sonderfälle eines unit, überdies sind *nonref* und *refmod* beide eine Möglichkeit des (allgemeinen) *amode*.

So gibt es (abgesehen von kollateralen Vereinbarung mehrerer identifiers) nur eine identity-declaration und diese hat die allgemeine Form:

amode identifier = unit

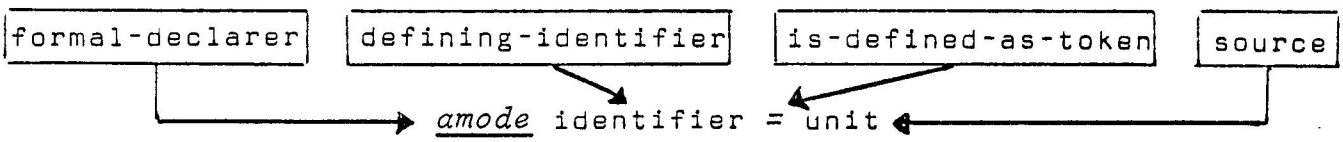
Es kommt noch dazu dasz die variable-declaration und die procedure-declaration semantisch in allen Hinsichten gleichwertig mit bestimmten identity-declarations sind. Sie bestehen nur um bequeme Abkürzungen für die Programmierpraxis zu ermöglichen (die variable-declaration ist auszerdem auch schon in dieser Form von ALGOL60 her bekannt, das *heap* *amode* identifier ist die verbesserte Fassung des ALGOL60-'*own*'-Konzepts).

Zusammenfassend kann man also sagen: es gibt in ALGOL68 nur eine identifier-declaration, die oben eingerahmte identity-declaration.

Die identity-declaration ist das weitaus wichtigste Konzept der Sprache; wer sie einmal ergründet hat, der versteht ALGOL68

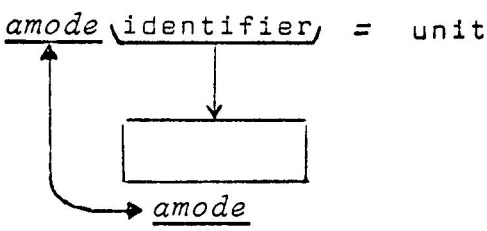
3.1.9. Die "elaboration" einer identity-declaration

Die syntaktische Komponenten der identity-declaration sind:



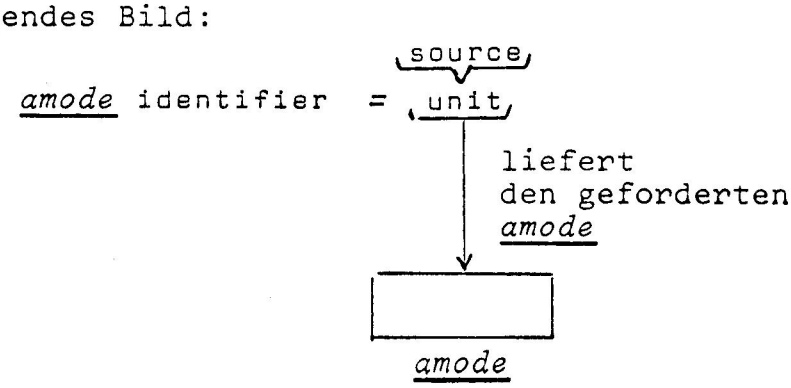
Der formal-declarer ist dabei entweder ein mode-indication oder ein mit Hilfe von 'mode-makers' aus mode-indications zusammengesetzter declarer. Ein declarer spezifiziert stets einen "mode".

Der 'compile-time'-Effekt einer identity-declaration ist folgender: durch den formal-declarer wird der "mode" des Wertes, zu welchem der defining-identifier Zugriff ("access") haben soll, festgelegt:

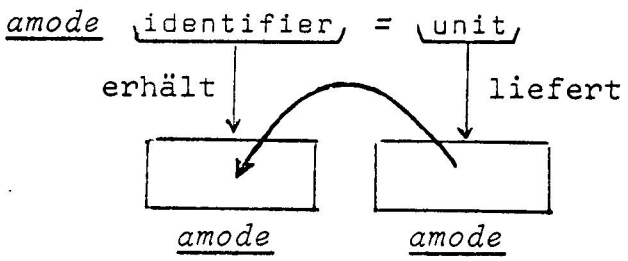


Der Compiler kennt also stets den "mode" des Wertes jedes defining-identifier und (wegen der Identifizierung, vgl. 2.4.2) den "mode" aller applied-identifier.

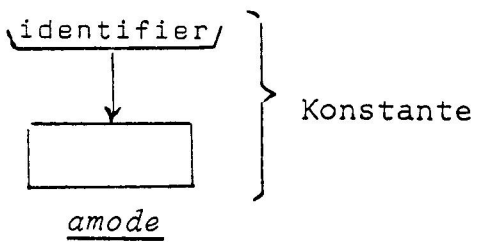
Interessanter ist der 'runtime'-Effekt der identity-declaration: Die source soll einen *amode*-Wert liefern. Wir wissen schon, dasz ein unit immer einen Wert (einschlieszlich *void*) erhält. Weil nun der Compiler den "mode" von den Werten aller applied-identifiers kennt, so kann er stets feststellen, ob die source (das unit rechts von =) einen *amode*-Wert liefern kann, ob also die identity-declaration syntaktisch korrekt ist.
Zur 'runtime' wird dann die source so abgearbeitet, dasz sie tatsächlich einen *amode*-Wert liefert. Dabei sind (wie wir später sehen werden, vgl. 4.4.2) alle "coercions" (vgl. 2.2.2) erlaubt. Wir haben also folgendes Bild:



Das Ergebnis der "elaboration" einer identity-declaration ist dann:

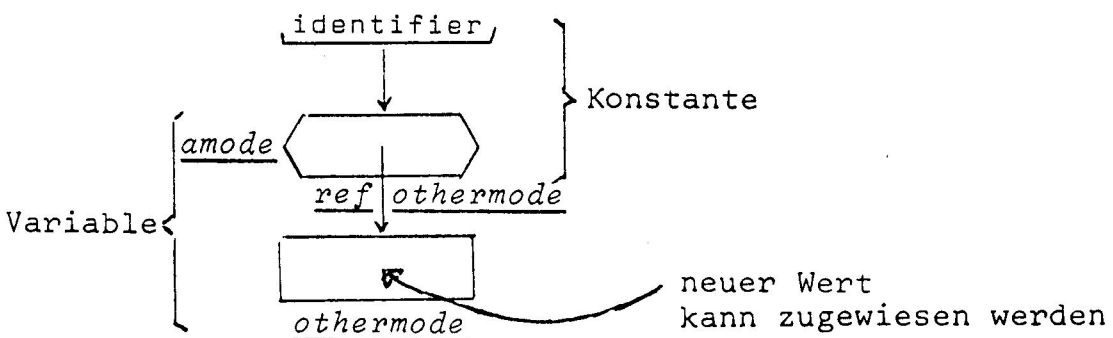


Nach der "elaboration" seiner identity-declaration hat also der defining-identifier (und wegen der Identifizierung auch alle applied-identifiers) Zugriff zu einem (durch die identity-declaration) bestimmten Wert. Dieser, dem identifier zugeordneten, Wert kann man nicht durch Zuweisung ändern: die Relation:



kennen wir bereits als eine Konstante (vgl. 1.2.5).

Man beachte jedoch, dasz *amode* sehr wohl ein *ref* *othermode* sein kann; dann ergibt sich folgendes Bild:



Auch hier kann der *amode* (*ref* *othermode*) nicht durch Zuweisung geändert werden: es wird an *amode*(*ref* *othermode*) zugewiesen, das *othermode*-"locale" erhält einen neuen Wert.

Nur durch (wiederholte) "elaboration" seiner identity-declaration kann der defining-identifier "access" zu einem neuen Wert erhalten.

Im folgenden unterscheiden wir den Fällen *mode* *amode* = *nonref* und *mode* *amode* = *refmod* lediglich auf didaktischen Gründen.

3.2. *nonref*-declarations

3.2.1. Deklaration von *nonref*-identifier

Die allgemeine Form die wir jetzt betrachten ist:

nonref identifier = unit

Einfache (und nützliche) Beispiele sind:

```
int max int = 4294967295 ;

real pi     = 3.1415926536 ,
     e      = 2.7182818285 ,
     gamma  = 0.5772156649 ;

compl i     = (0.0,1.0)
```

Man beachte dasz die identifiers *maxint*, *pi*, *e*, *gamma* und *i* hierdurch als Konstanten definiert worden sind (*maxint* und *pi* sind, implementations-abhängig, standarmäszig vorgegeben). Eine Zuweisung an solche Konstanten ist also unmöglich:

pi := 2.7182818285

ist syntaktisch inkorrekt und, abgesehen davon, genau so unsinnig wie zum Beispiel:

3.1415926536 := 2.7182818285

In eine gute Implementation darf man erwarten dasz der Zugriff zu solchen Konstanten genau so effizient geschieht als durch die entsprechende denotations. Der *compl i = (0.0,1.0)* ist ganz hübsch weil man durch ihn neben der 'offizielle' Verknüpfung zweier *real*s zu einem *compl*:

z := u i v

die mathematisch vertrautere Schreibweise

z := u + i×v

verwenden kann.

Weitere nützliche Konstanten sind:

```
real ln10 = ln(10) , sqrt2 = sqrt(2) , sqrtpi = sqrt(pi) ,
     lnpi = ln(pi) , sqrt2pi = sqrt(2×pi) ,
     planck = 6.6254 -27 , avogadro = 6.0247 23 ;
                10          10
```

Auch in anderen Programmiersprachen versucht man natürlich alle ~~xxx~~ Gröszen die man in seinem Programm (oder Programmteil) des öfteren braucht, durch Zuweisung an eine Variable nur einmal zu berechnen. So wird man in ALGOL60 z.B. schreiben:

```
real sqrt2pi; sqrt2pi:=sqrt(2×pi);
```

In ALGOL68 ist jedoch die Benutzung einer ('lokalen') *nonref*-identity-declaration noch effizienter weil man zu dem gewünschten Wert direkt (und nicht nur über einen *ref real*) zugreifen kann. Beispiel:

```
for i to n
do real u = row(i);
   for j to n
   do real v = row(j) ;
      real gausz = u×v + exp(-(u-v)↑2/2)/sqrt2pi;
      serial-clause
   od
od
```

Noch zwingender wird diese Überlegung wenn man des öfteren eine Kopie der i-ten Zeile (j-ten Spalte) eines "multiple value" braucht.

Wie wir wissen, spezifiziert ein formal-declarer einen "mode". Ebenso wissen wir, dasz bei dem "mode" eines "multiple value" die Grenzen nicht betrachtet werden, man macht also nur einen Unterschied zwischen

```
[]amode , [,]amode , [][]amode ,
[][,]amode , [,][]amode , [,,]amode und so weiter
```

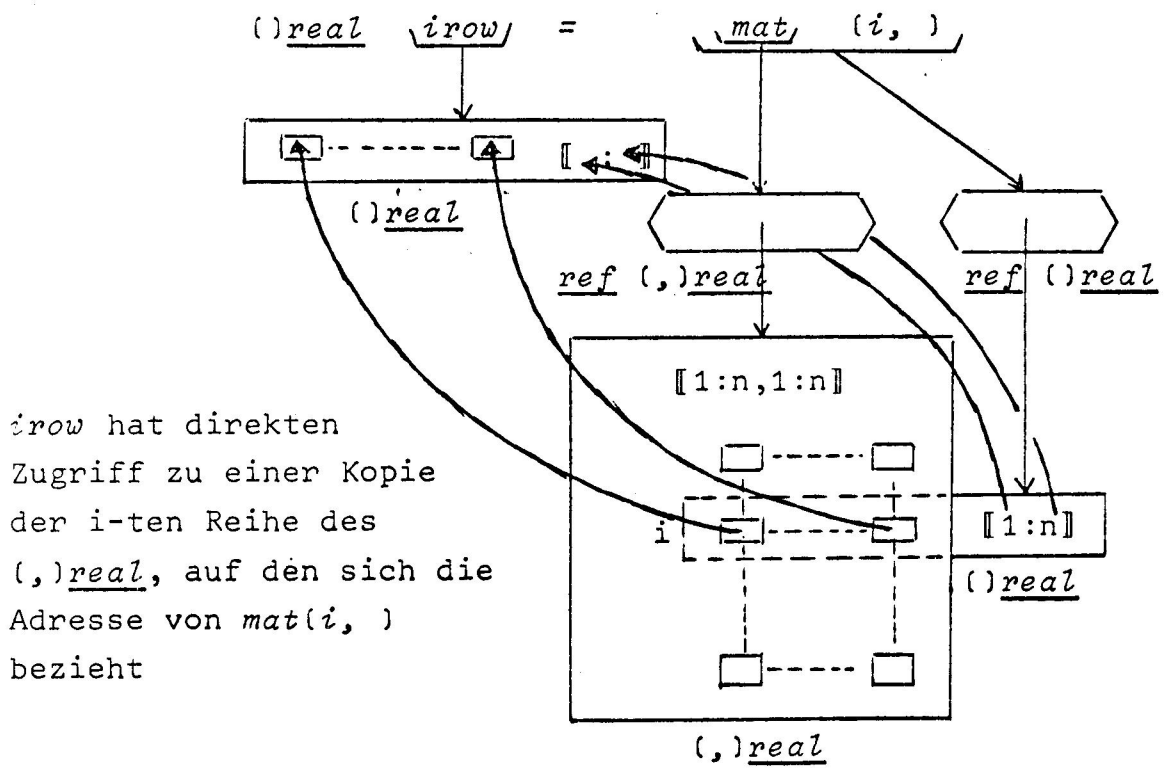
Regel: in einem formal-declarer spielen die (aktuelle) Grenzen keine Rolle

Will man dann mit Hilfe eines formal-declarer den "mode" eines "multiple value" spezifizieren, so sind Grenzen innerhalb dieses formal-declarer, da sie semantisch keinen Sinn haben können, syntaktisch von vornherein nicht gestattet.

Man schreibt also, zum Beispiel:

```
[]real irow = mat(i, )
```

Die "elaboration" dieser identity-declaration ergibt dann folgendes Bild:



Man hat nun z.B. durch *irow*(*j*) einen weitaus effizienteren Zugriff zu dem Element (i,j) als durch *mat*(*i*,*j*), da nur ein Paar von Grenzen überprüft zu werden braucht und bei der Indexberechnung keine Multiplikation mehr stattfindet.

Es wird im allgemeinen empfohlen, einen identifier immer dann als Konstante (mit Hilfe einer *nonref*-identity-declaration) zu definieren, wenn an ihn kein Wert zugewiesen wird.

In diesem Zusammenhang ist es interessant noch einmal die loop-clause zu betrachten (vgl. 2.7):

```
for control-identifier from start-unit
                       by step- unit
                       to stop- unit
                    while enquiry-clause
do serial-clause od
```

Wir konstruieren nun eine closed-clause die, so effizient wie möglich, in ALGOL68 die Semantik der loop-clause definiert.

Wir nehmen an, der Programmierer wählt als control-identifier *i*; seine loop-clause fängt also an mit *for i*. Wir wissen schon dasz dann dieser *i* auftritt als eine ganzzahlige Konstante: *i* ist ein identifier der stets ein *int* erhält (vgl. 2.7).

In der folgende closed-clause deklarieren wir sechs identifiers die wir mit Hauptbuchstaben schreiben; auf dieser Weise geben wir an dasz diese identifiers dem Programmierer nicht zur Verfügung stehen (er kennt sie sogar nicht) und verschieden sind von allen anderen identifiers in seinem Programm.

```
begin   int START  =  start unit  ,
            STEP   =  step  unit  ,
            STOP   =  stop  unit  ;
        int COVAR :=  START - STEP ;
       bool INCREASING STEP = STEP ≥ 0 ;
REPEATE:    (  int i = COVAR plusab STEP
                       co i is the control-identifier
                          after for
                       co ;
                if  if INCREASING STEP
                  then i ≤ STOP else i ≥ STOP
                    fi
              then  if enquiry clause
                       co this is the enquiry-clause
                          after while
                       co
                  then serial clause
                       co this is the serial-clause
                          between do and od
                       co ;
                       goto REPEATE
                    fi
                fi
            )
end
```

3.2.2. Deklaration von Prozeduren

Auch eine Prozedur kann man mit Hilfe einer identity-declaration vereinbaren. Will man zum Beispiel erreichen dasz der identifier *sinx* die Routine, bestimmt durch den routine-text:

(real x) real: sin(x)/x

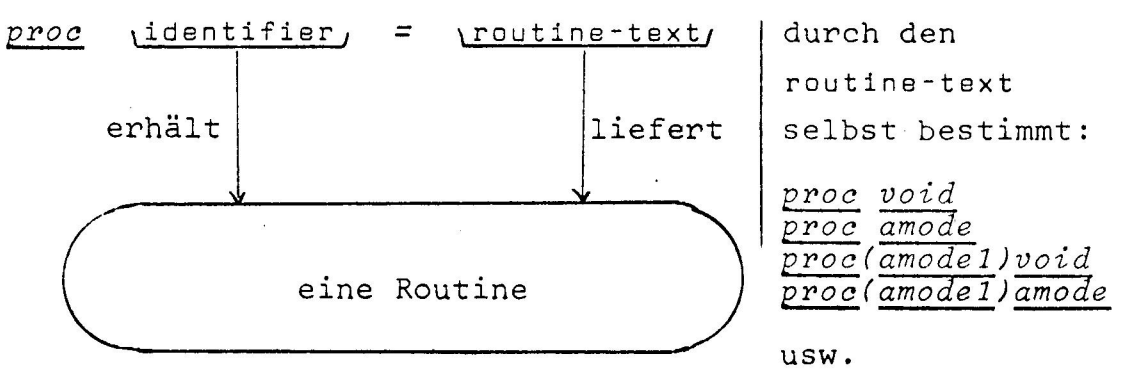
erhält, so kann man schreiben:

proc(real)real sinx = (real x)real: sin(x)/x

Der formal-declarer ist hier also *proc(real)real* und der source ist ein routine-text der eine Routine des gewünschten "mode" erhält. Ein routine-text beschreibt jedoch bereits selbst genau den "mode" seines Wertes (eine ganz bestimmte Routine). Es ergibt sich also dasz (in diesem Fall) der formal-declarer eigentlich überflüssig ist, das symbol "*proc*" allein würde hier schon genügen.
Wenn nun, in einer identity-declaration, der source ein routine-text ist, der formal-declarer also nur einen ganz bestimmten *procmode* spezifizieren kann, so gestattet die Syntax hier eine procedure-declaration. Man kann dann abkürzend schreiben:

proc sinx = (real x)real: sin(x)/x

Ihre "elaboration" ist (für die Programmier-Praxis) semantisch völlig gleichwertig mit der entsprechenden identity-declaration. Wir erhalten folgendes Bild:



Man beachte dasz die procedure-declaration nur dann erlaubt ist, wenn in einer *procmode*-identity-declaration der source ein routine-text ist (vgl. 3.2.3).

Als Beispiel betrachten wir jetzt eine kleine library von Prozeduren:

Für ein Textbehandlungsprogramm will man einige handliche Prozeduren, die *char* nach *char* die Eingabe abtasten bis ein bestimmter *char* gefunden ist, zur Verfügung haben. Man kann dann deklarieren:

a) ein *proc char* :

```
proc next        = char: (char sym; read(sym); sym) ;
```

b) ein *proc void* :

```
proc findblank   = void:  while next ≠ " "
                          do skip od ;
```

c) ein *proc char* :

```
proc nexttoblank = char: (findblank; next) ;
```

d) ein *proc(char)void* :

```
proc find        = (char stop)void: while next ≠ stop
                                    do skip od ;
```

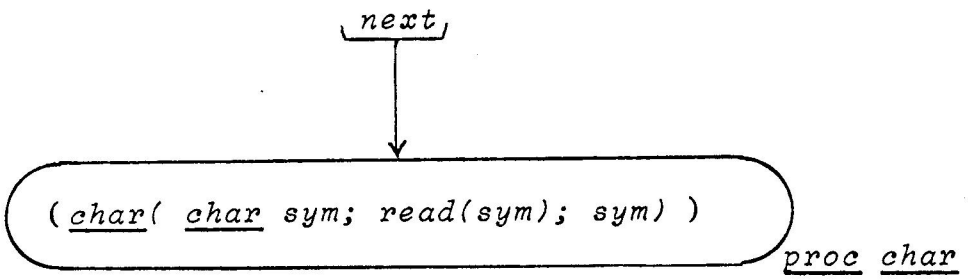
e) ein *proc(char)char* :

```
proc nextto      = (char stop)char: (find(stop); next) ;
```

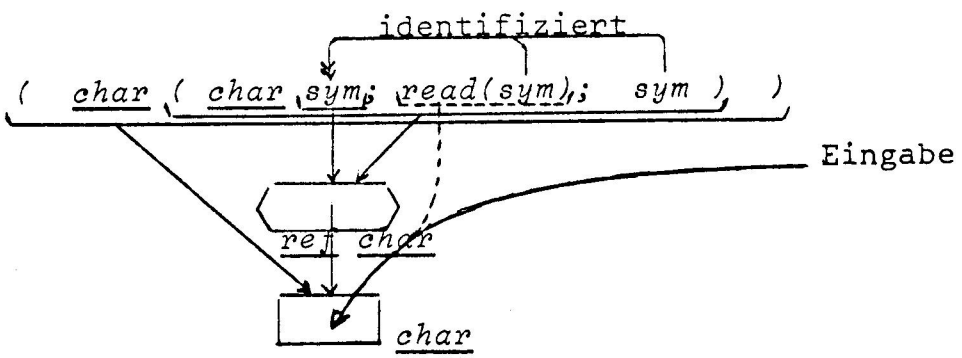
Wie man diese Prozeduren aufruft und, ins besondere, wie bei c) und d) die aktuellen Parameter übertragen werden, wird in den folgenden Paragraphen behandelt.

3.2.3. Prozeduraufruf (ohne Parameter)

Durch die procedure-declaration a) im vorigen Paragraphen hat *next* Zugriff zu einer Routine vom "mode" *proc char* :

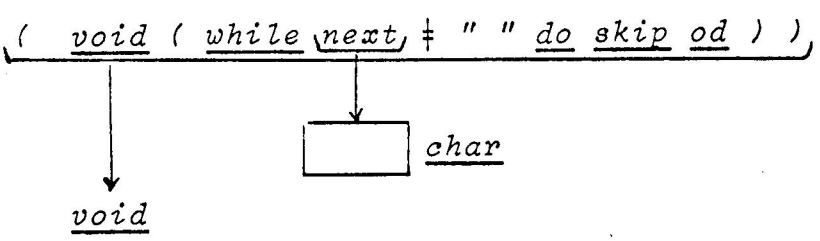


Der Aufruf der Prozedur *next* geschieht durch Aufruf ihrer Routine, d.h. eine "action" die semantisch gleichwertig ist mit der "elaboration" der closed-clause *(char(char sym; read(sym); sym))*.
Wir erhalten also folgendes Bild:



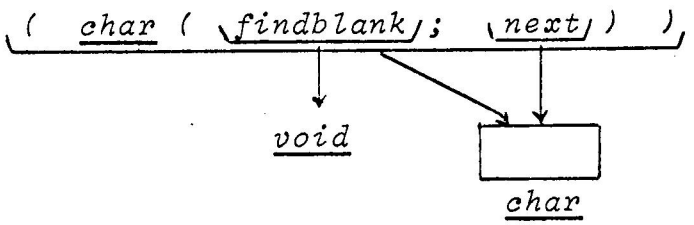
Jeder Aufruf von *next* liefert also den nächsten *char* auf dem Eingabemedium.

Ein Aufruf von *next* findet z.B. bei dem Aufruf der Prozedur *findblank* statt; denn der Aufruf von *findblank* geschieht durch die "elaboration" der closed-clause:



Bei dem Aufruf von *findblank* wird *next* so oft aufgerufen bis ein 'blank' " " geliefert wird; der Aufruf von *findblank* liefert dagegen selbst ein *void*.

Sowohl ein Aufruf von *findblank* als auch von *next* findet beim Aufruf von *nexttoblank* statt. Dies geschieht durch die "elaboration" der closed-clause:



Durch den Aufruf von *nexttoblank* wird also der dem gefundenen 'blank' folgende *char* (auf dem Eingabemedium) geliefert.

Zusammenfassend:

> der Aufruf einer *proc void* bzw. einer *proc amode* ist die "elaboration" einer closed-clause die durch leichte Abänderung aus dem entsprechenden routine-text entsteht (vgl. a und b in 3.1.5).

Wichtig ist noch dabei, dasz die "elaboration" dieser closed-clause stattfindet in der Umgebung der procedure-declaration (also nicht in der Umgebung des Aufrufs). Wir betrachten folgendes Beispiel:

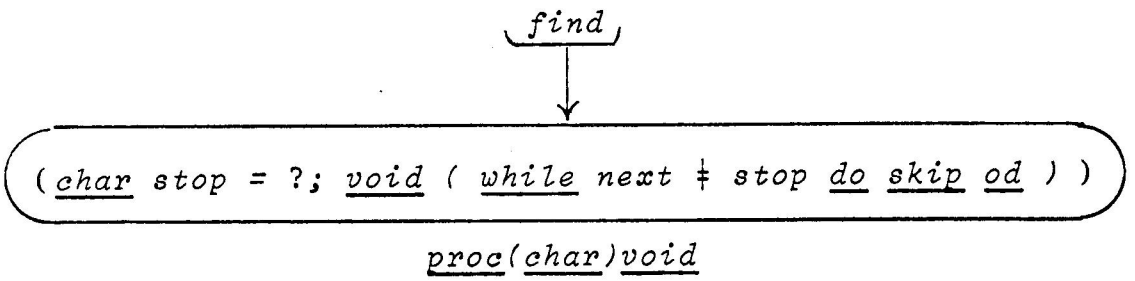
```
begin   int count := 0  co this is count1 co ;
        proc nextcount = char: (char sym; read(sym);
                                count plusab 1; sym) ;
         serial-clause ;
         begin   int count := 0  co this is count2 co ;
                 char symbol := nextcount ;
                  serial-clause
         end ;
         serial-clause
end
```

identifiziert

Der Aufruf von *nextcount* geschieht durch die "elaboration" der closed-clause *(char(char sym; read(sym); count plusab 1; sym))* in der Umgebung ("range") worin *count1* gilt. Der Aufruf erhöht also *count1* und ändert nicht *count2* (!).

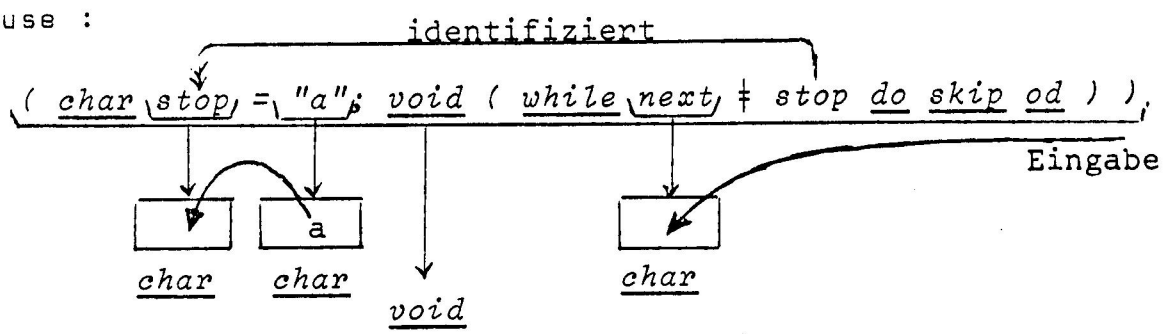
3.2.4. Prozeduraufruf ('call by value')

Durch die procedure-declaration c) von 3.2.2 hat *find* Zugriff zu einer Routine vom "mode" *proc(char)void* :



Betrachten wir nun den call *find("a")*.

Die Parameterübergabe geschieht so, alsob an die Stelle von "?" der aktuelle Parameter *"a"* tritt. Die "elaboration" des call *find("a")* geschieht danach durch die "elaboration" der closed-clause :

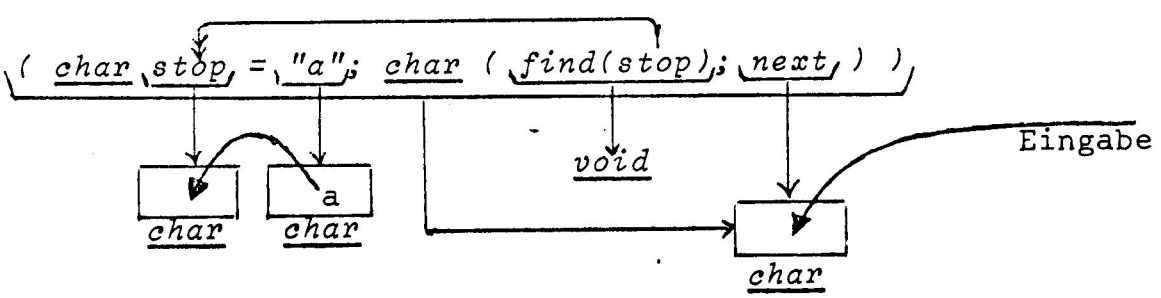


Es ist also die identity-declaration :

char stop = "a"

die die Parameterübergabe vornimmt. Innerhalb der closed-clause vertretet *stop* den aktuellen Wert *"a"*; in der formula *next ≠ stop* wird also den erstfolgenden *char* (auf dem Eingabemedium) verglichen mit *stop*, also mit dem aktuell übertragenen Wert *"a"*.

Ein Aufruf von *find* findet z.B. bei der "elaboration" des call *nextto("a")* statt, also durch die "elaboration" der closed-clause :



Die "elaboration" von *nextto("a")* liefert demnach den *char*, der auf dem Eingabemedium dem zuerst gefundenen *"a"* folgt. Man beachte wie hier den aktuellen Wert *"a"* zunächst an *nextto* übertragen wird, und dann (bei der "elaboration" von *nextto("a")*) noch einmal an *find*.

Auch hier werden die zu *nextto* bzw. *find* gehörende closed-clauses in der Umgebung ("range") der procedure-declarations abgearbeitet. Der aktuelle Parameter jedoch, wird berechnet (abgearbeitet) in der Umgebung ("range") des call.

Es lohnt die Mühe folgendes Beispiel mal gut zu ergründen:

```
begin   int count := 0 ; co this is count1 co
        proc findcount = (char stop)void:
                         while next ≠ stop
                            do count plusab 1 od
        serial-clause ;
        begin   int count ; co this is count2 co
                serial-clause ;
                find ( repr ( abs"a" + count ) ) ;
                                  char
                serial-clause ;
        end
        serial-clause
end
```

Der aktuelle Parameter *repr(abs"a"+count)* wird abgearbeitet in der Umgebung von *count2*, die "elaboration" der zu *find* gehörenden closed-clause dagegen findet statt in der Umgebung von *count1*.

Wenn der formal-declarer einen *nonref* fordert (wie es in obigen Beispiele der Fall war), so kann man (wie in anderen Programmiersprachen) von einem 'call by value' sprechen.

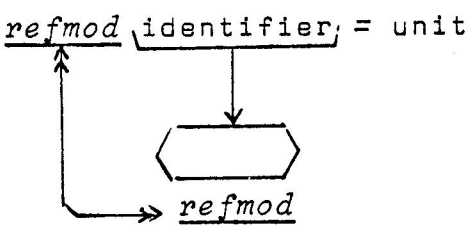
3.3. *refmod*-declarations

3.3.1. Deklaration von *refmod*-identifiers

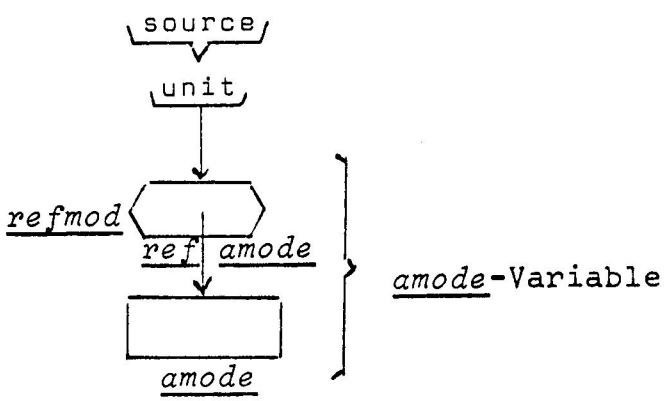
Die allgemeine Form die wir jetzt betrachten ist:

refmod identifier = unit

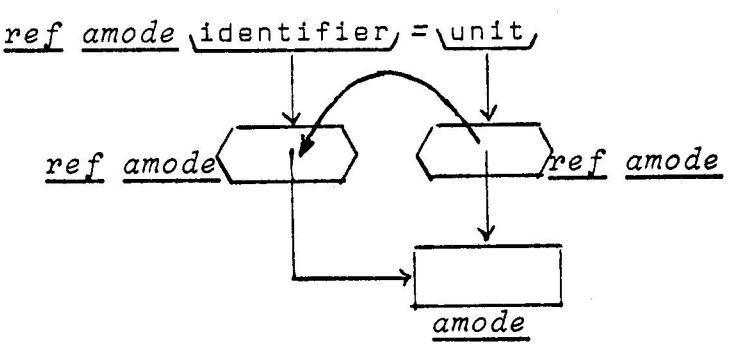
Der 'compiletime'-Effekt dieser identity-declaration ist folgender: durch die linke Seite hat der defining-identifier Zugriff ("access") zu einem *refmod*, also zu einer Adresse, die sich auf einen anderen Wert beziehen kann (vgl. 3.1.9):



Die source soll hier also einen *refmod*-Wert liefern, d.h. eine Adresse, die sich tatsächlich auf einen anderen Wert bezieht (also eine Variable):

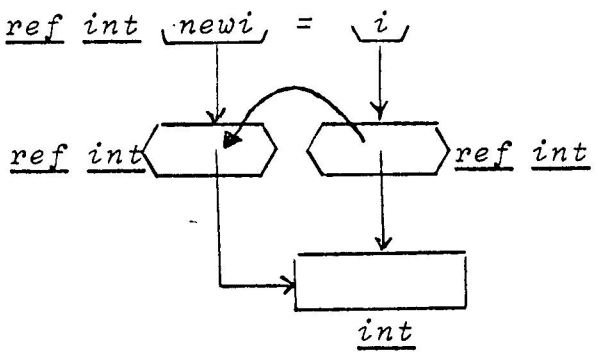


Die "elaboration" einer *refmod*-(*ref* *amode*) identity-declaration liefert also folgendes Bild:



Der Erfolg einer *refmod*-identity-declaration ist sehr bemerkenswert: nach ihrer "elaboration" hat der defining-identifier Zugriff ("access") zu der Adresse eines anderen Wertes im Speicher, d.h. zu einer Variable.

Beispiel:



Man beachte hierbei, wenn man im folgenden der Adresse von *i* einen *int*-Wert zuweist, man gleichzeitig diesen *int*-Wert auch der Adresse zu der *newi* Zugriff hat, zugewiesen hat.

In FORTRAN und verwandten Sprachen wird so ein Sachverhalt wohl 'equivalence' genannt: verschiedene identifiers mit gleichen Bedeutung (ALGOL60 kennt diese möglichkeit nicht in dieser expliziter Form).

Man beachte auch, wie dieser auffallender Sachverhalt unmittelbar erfolgt aus dem allgemeinen Konzept der identity-declaration und seine "elaboration" (es liegt hier kein neues Konzept vor, vgl. 3.1.9).
Man kann sogar sagen: ALGOL68 kennt nur eine identity-declaration: die declaration einer Konstante.
In:

int new0 = 0

erhalten *new0* und *0* beide Zugriff zu den gleichen *int*-Wert,
in :

ref int newi = i

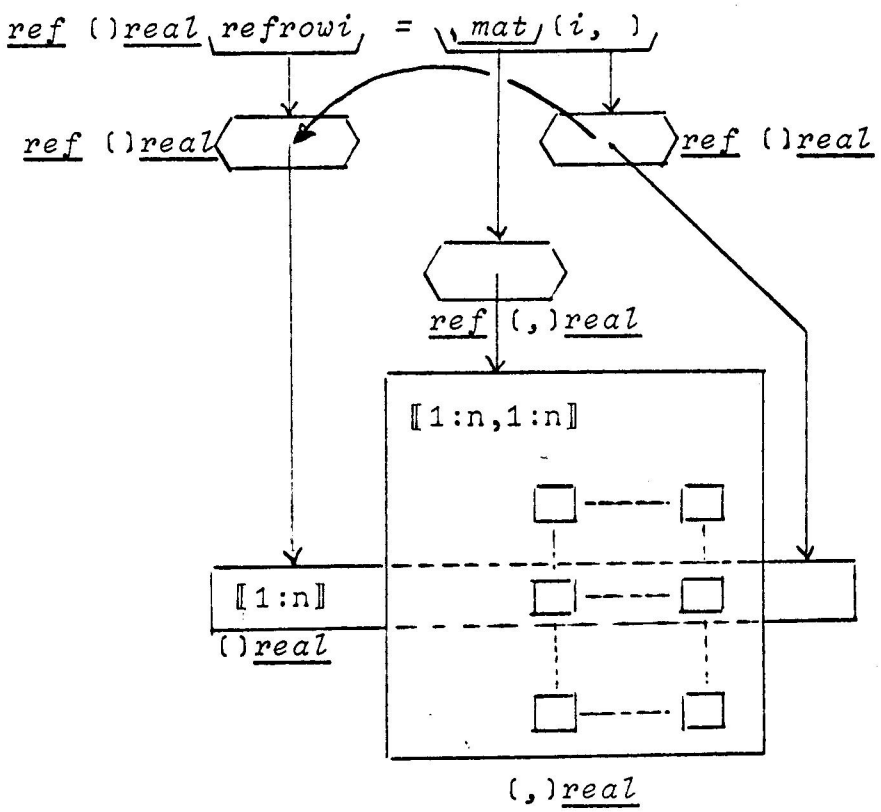
erhalten *newi* und *i* beide Zugriff zu den gleichen *ref int*-Wert.
Blosz weil ein *ref amode*-Wert sich bezieht auf einen *amode*, sind die Konsequenzen der *ref amode*-identity-declaration so weitreichend.

In 3.2.1 betrachteten wir die identity-declaration

()*real* *irow* = *mat*(*i*,)

Hierdurch erhält *irow* eine Kopie der i-ten Reihe der *matrix*, auf die sich die Adresse von *mat* bezieht. Dies ist z.B. immer dann nützlich, wenn man zu einem bestimmten Zeitpunkt die i-te Reihe retten will, da man durch nachfolgenden Zuweisungen gerade diese i-te Reihe ändern will.

Sehr oft benötigt man jedoch nur die Adresse, die sich auf die i-te Reihe bezieht, man also die einzelnen Elemente nicht zu kopieren braucht (man spart also Speicherplatz). Dann kann man schreiben:



Man beachte dasz hier z.B. *refrowi*(*j*) und *mat*(*i*,*j*) beide zu der Adresse Zugriff haben, die sich auf das Element (*i*,*j*) bezieht; durch *refrowi*(*j*) braucht man nur eine Indizierung, dieser Zugriff ist also viel effizienter als durch *mat*(*i*,*j*) (vgl. auch 3.2.1).

Man vergleiche: durch ()*real* *irow* = *mat*(*i*,) erhält *irow*, ohne Vermittlung eines *ref* Zugriff zu einer Kopie der i-ten Reihe von *mat*; durch *ref* ()*real* *refrowi* = *mat*(*i*,) erhält *refrowi* Zugriff zu der Adresse der i-ten Reihe, die Reihe selbst wird nicht kopiert.

3.3.2. Prozeduraufruf ('call by reference')

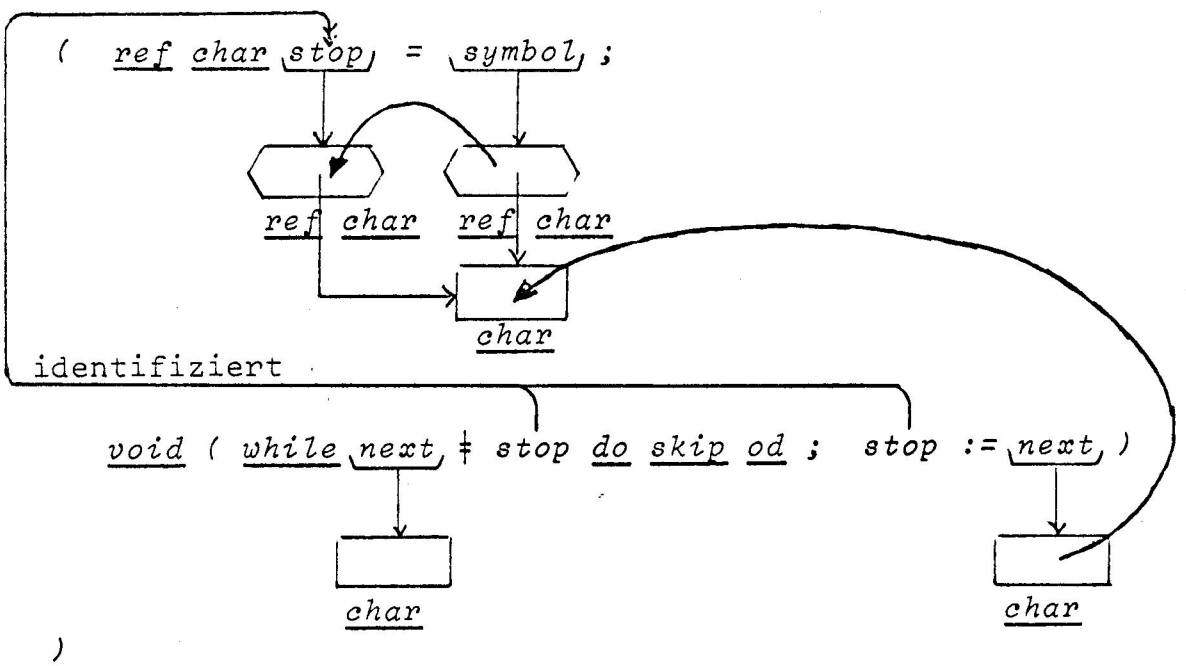
Innerhalb einer procedure-declaration kann der formal-declarer in dem routine-text selbstverständlich auch einen *ref amode* spezifizieren.
Man hat damit die Möglichkeit, innerhalb der so definierten Routine über den formalen Parameter einer auszerhalb der Routine deklarierten Variablen einen neuen *amode*-Wert zuzweisen.

Beispiel:

ein *proc(ref char)void* (vgl. 3.2.2):

```
proc findput = (ref char stop)void:
                    ( while next ≠ stop do skip od;
                      stop:=next )
```

Die "elaboration" z.B. eines call *findput(symbol)* geschieht sodann durch die "elaboration" der closed-clause (vgl. auch 3.2.4):



Man beachte hierbei, wie die assignation *stop:=next* durch *stop* den (beim Aufruf von *next* gelieferten) *char* zuweist an den aktuellen Parameter *symbol*. Innerhalb der Routine vertretet die Variable *stop* die aktuelle Variable *symbol*, diese Vertretung wurde zustande gebracht durch die identity-declaration

ref char stop = symbol

Will man einer Routine ein ()*amode* (also ein "multiple value") übergeben, so ist es in fast allen Fällen angebracht, den formal-declarer der Routine so anzugeben, dasz er einen *ref* ()*amode* und nicht einen ()*amode* spezifiziert. Es werden dann bei der Übergabe nicht die Elemente des "multiple value" kopiert, sondern nur eine einzige Adresse (nämlich die, die sich auf das "multiple value" bezieht).

So ist es also angebracht in dem Beispiel {7} von 1.4.6 statt

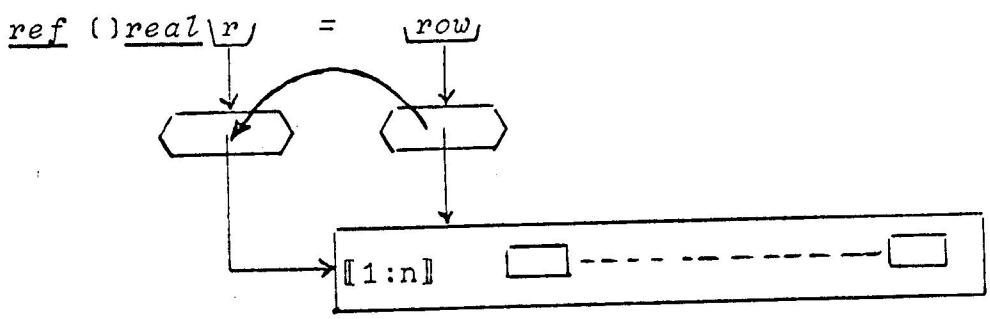
```
proc compute = (()real r)quadr: ( serial-clause )
```

besser

```
proc bettercompute = (ref ()real r)quadr: ( serial-clause )
```

zu schreiben.

Man beachte, dasz der Effekt der beiden Prozeduren der gleiche ist (sogar ohne Änderung des 'Prozedur-Rumpf'), er wird jedoch durch *bettercompute* weitaus effizienter erzielt: bei der Parameter-Übergabe wird nicht die ganze Reihe kopiert, nur seine Adresse:



Noch ein Beispiel:

```
proc max = (ref ()real r, ref int imax) real:
                ( for i
                 from (imax:=lwb r) + 1
                   to upb r
                   do   if r(i)>r(imax)
                      then imax:=i
                        fi
                   od ;
                   r(imax)
                )
```

Eine nützliche Anwendung eines *refmod*-formal-declarer ist noch der *proc(ref int)real nextrandom*, der in der standard-prelude wie folgt definiert ist (dabei gibt *C* ein besonderes "comment" an, wofür man sich Objektcode denken kann der gerade dasjenige leistet das in diesem "comment" in Wörter beschrieben wird):

```
proc nextrandom = (ref int a) real:
    ( a := C the next pseudo-random integral value after a
             from a uniformly distributed sequence on the
             interval ( 0 , max int )
          C ;
          C the real value corresponding to a according to
             some mapping of integral values ( 0 , max int )
             into real values ( 0 , 1 ), i.e. such that
             0 ≤ x < 1 , such that the sequence of real values
             produced preserves the properties of pseudo-
             randomness and uniform distribution of the
             sequence of integral values
          C
    )
```

Für die Verwendung von *nextrandom* braucht man also eine *int*-Variable, zum Beispiel:

```
int lastrandom; read(lastrandom);
```

Bei jedem Aufruf von *nextrandom* wird nun *lastrandom* geänderd und *lastrandom* durchlauft dabei eine stochastische Folge ganzer Zahlen; *nextrandom(lastrandom)* erhält dann einen entsprechenden *real*-Wert zwischen *0* und *1* (zum Beispiel: *lastrandom/maxint*). So kann man u.a. nach jedem Aufruf aus dem Wert von *lastrandom* Würfelzahlen bilden durch *lastrandom mod 6*.
Für eine Anwendung von *nextrandom* vgl. 3.3.3.

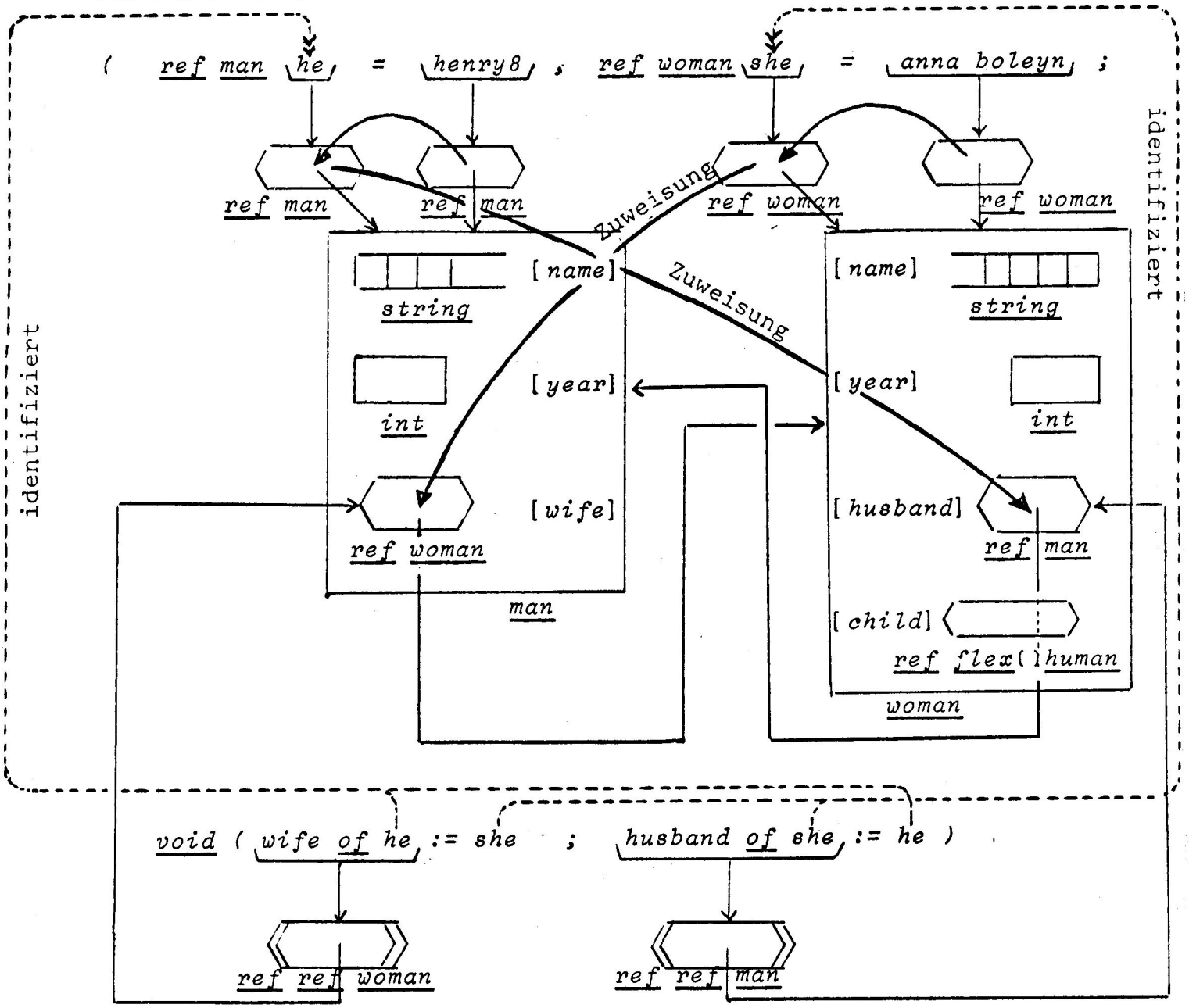
Auch die Trauung können wir nun in allen Einzelheiten verstehen:

```
proc marry = (ref man he, ref woman she) void:
                 (   wife of  he := she ;
                  husband of she :=  he
                 )
```

Wir betrachten die Heirat:

```
marry ( henry8 , anna boleyn )
```

also die "elaboration" der closed-clause :



)

3.3.3. Routinen die einen *refmod*-Wert liefern

Im vorigen Paragraphen haben wir gesehen, wie der formal-declarer einen *refmod* spezifizieren kann. Natürlich kann man auch eine solche Prozedur deklarieren, deren Routine bei ihren Aufruf einen *refmod* liefert.

Beispiel:

```
proc x or y = ref real:   if nextrandom(n) < 0.5
                        then x
                        else y
                          fi
```

Die Prozedur *xory* erhält also Zugriff zu einem *proc ref real*, d.h. einer Routine ohne Parameter, die bei ihrem Aufruf als Ergebnis ein *ref real* liefert.

Wir können dan einen Aufruf als linke Seite einer assignation schreiben, etwa:

```
x or y := sqrt(pi)
```

Hierdurch wird also entweder der Adresse von *x* oder der Adresse von *y* die Quadratwurzel aus *pi* zugewiesen.

Beispiel:

Es sei deklariert:

```
real one, two, three, four, five, six;
int dice ;
```

```
proc throw = (ref int n) ref real:
                case nextrandom(n); n mod 6 + 1
                  in one , two , three , four , five , six
                esac ;
```

Auch hier können wir den Aufruf als linke Seite einer assignation schreiben:

```
throw(dice) := sqrt(pi)
```

Man kann sogar (syntaktisch korrekt) schreiben:

```
throw(dice) := x or y
```

oder auch

```
x or y := throw(dice)
```

Man erinnere sich jedoch dasz in einer assignation die destination und der source kollateral abgearbeitet werden. Man wiesz also nicht welcher Aufruf von *nextrandom* erst kommt (sogar nicht ob *nextrandom* überhaupt zweimal aufgerufen wird); es kann aber sein dasz diese Unsicherheit gerade bei stochastischen Auswahlen nicht stört.

Als Schluszbeispiel betrachten wir noch folgendes:

Es sei deklariert:

```
int n; read(n); (1:n)real myrow; read(myrow);
```

Wir wollen nun eine Prozedur *max* deklarieren, die die Adresse des gröszen Elementes einer ()*real* liefert und überdies einem aktuell mitzugebenen *ref int* den "subscript" dieses gröszten Elementes zuweist, man dann also zum Beispiel schreiben kann:

```
max(myrow,k) := 0
```

Nach dieser assignation weisz man dann durch *k* noch welchem Element man den Wert *0* zugewiesen hat (also welches Element das gröszte war). Die Prozedur *max* musz also durch ihre declaration eine Routine erhalten, die bei ihrem Aufruf ein *ref real* liefert:

```
proc max = (ref ()real row, ref int imax) ref real:
                ( imax := lwb row;
                  for i from imax+1 to upb row
                  do
                      if row(i)>row(imax) then imax:=i fi
                  od ;
                  row(imax)
                )
```

In dieser Routine wird innerhalb der loop-clause bei jedem Schritt der Vergleich *row(i)>row(imax)* durchgeführt. Dabei wird also jedesmal zweimal (also 2nx) durch Indizierung ein Element ausgewählt. Da die Auswahl von Elementen durch Indizierung recht zeitaufwendig sein kann, sollte man sie dort vermeiden, wo sie nicht unbedingt erforderlich ist.
Bei der Suche nach dem gröszten Element aus einer Reihe kommt man durchaus mit weniger Fällen, bei denen Elemente durch Indizierung ausgewählt werden, aus als in dem angegebenen routine-text.

Wenn wir annehmen dasz die Reihe nicht schon (zufällig) in aufsteigender Folge geordnet war, dasz also das Ereignis dasz ein gröszeres Element gefunden wird nicht zu oft auftritt, liefert der folgende routine-text eine Routine die die Suche effizienter (sei es weniger transparent) durchführt als die oben angegebene Routine.

```
proc max = (ref ()real row, ref int imax) ref real:
        ( int upbrow = upb row;
          imax := lwb row;
          nextloop:
            ( real rowimax = row(imax);
               for i
              from imax+1
                to upbrow
                do
                   if row(i)>rowimax
                then imax:=i; nextloop
                   fi
                od
            ) ;
          row(imax)
        )
```

4. DIE ANPASSUNGSOPERATIONEN (COERCIONS)

4.1 Einfache Anpassungsoperationen

4.1.1 Der Begriff "coercion"
4.1.2 "dereferencing"
4.1.3 "deproceduring"
4.1.4 "rowing"
4.1.5 "voiding"
4.1.6 Jump und *skip*

4.2 Verwandte "modes"

4.2.1 *bool* und *bits*
4.2.2 *char* und *bytes*
4.2.3 *long* und *short* "modes"
4.2.4 "widening"

4.3 "United-modes"

4.3.1 Der Begriff *union*
4.3.2 Die *union*-Variable
4.3.3 "Uniting" und Zuweisung an eine *union*-Variable
4.3.4 Die conformity-clause

4.4 Die Durchschlagskraft des Kontextes

4.4.1 Der Begriff der syntaktische Position
4.4.2 Die verschiedene Arten des Kontextes
4.4.3 Der cast

---

4. DIE ANPASSUNGSOPERATIONEN (COERCIONS)

4.1. Einfache Anpassungsoperationen

4.1.1. Der Begriff "coercion"

Durch eine "coercion" wird implizit ein internes Objekt eines bestimmten "mode" in ein internes Objekt eines anderen "mode" übergeführt (vergleiche jedoch 4.3.3).
In allen Programmiersprachen treten solche Arten von impliziten Typ- bzw. Wert-Umwandlungen auf.
In 2.2.2 betrachteten wir schon das Beispiel:

$$y := i \times j$$

Die "apriori-modes" von i und j sind *ref int*; der "apriori-mode" von $i \times j$ ist *int*; die destination y fordert jedoch einen *real*. Der Compiler 'versteht' diese assignation dann als ob man explizit geschrieben hätte:

$$y := \underline{real}(\underline{int}(i) \times \underline{int}(j))$$

Auch ein ALGOL60-Compiler übersetzt das <assignment statement> in dieser Weise, und auch wenn man in FORTRAN schreibt $Y = I \times J$, so kann man nichts anderes erwarten.
Dasz in den meisten Programmiersprachen ziemlich unbekümmert über solche impliziten Anpassungsoperationen hinweggesehen wird, findet seine Ursache darin, dasz es sich hierbei um ein einfaches und vor allem natürliches Konzept handelt.

Die orthogonale Struktur von ALGOL68, besonders seine Möglichkeiten, beliebig viele neue "modes" und Operatoren zu definieren, sowie seine weitreichende Manipulation mit zusamengesetzten Werten ("multiple-" und "structured-values") machen es unvermeidlich, bei der Behandlung solcher impliziten Anpassungsoperationen äuszerst genau zu sein. In ALGOL68 sind die "coercions" Bestandteile der Syntax, d.h. diese impliziten Anpassungsoperationen werden völlig durch den Kontext bestimmt. Gerade dies macht die Syntax recht kompliziert, die Anwendung der Sprache aber einfach und angenehm. Der Programmierer sollte jedoch ungefähr wissen, wann welche Anpassungsoperationen implizit angewendet werden.

Zunächst ein wenig Terminologie:

"to coerce" bedeutet: "anpassen" ,
"coercion" bedeutet: "Anpassung" ,
ein "coercer" ist ein "Anpasser" ,
ein "coercend" ist ein "Anzupassender" .

Der "coercer" ist in ALGOL68 immer der syntaktische Kontext; der "coercend" ein unit, dessen Wert in einen anderen (angepaszten) Wert übergeführt werden soll.

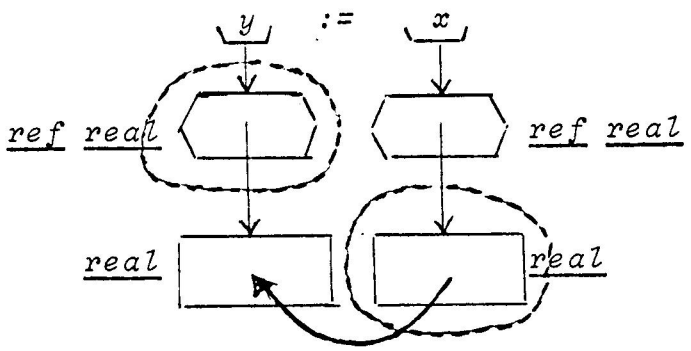
Ein unit erhält einen Wert eines bestimmten ("apriori-") "mode". Es kann durch seinen syntaktischen Kontext gezwungen werden, einen Wert eines anderen ("aposteriori-") "mode" zu liefern.

Wir werden die impliziten Anpassungsoperationen, wenn es zum besseren Verständnis nötig ist, explizit durch 'Compilations-Operatoren' angeben.
Die sechs Grund-"coercions" in ALGOL68 sind:

offizielle Terminologie	deutsche Übersetzung	Compilations-Operatoren
"dereferencing"	"absteigen" (von einer *ref amode* Adresse zu ihrem *amode* Wert)	deref
"deproceduring"	(eine "routine", die keine Parameter hat) 'aufrufen'	deproc
"rowing"	"zu einer Reihe machen"	row
"voiding"	(einen Wert) "nicht weiter betrachten"	voiden
"widening"	"erweitern"	widentoreal widentocompl widentorowofbool widentorowofchar
"uniting"	"vereinigen"	unite

4.1.2. "dereferencing"

Durch das "dereferencing" geschieht der Übergang von *ref amode* zu *amode*.
Wir betrachten folgende assignation:

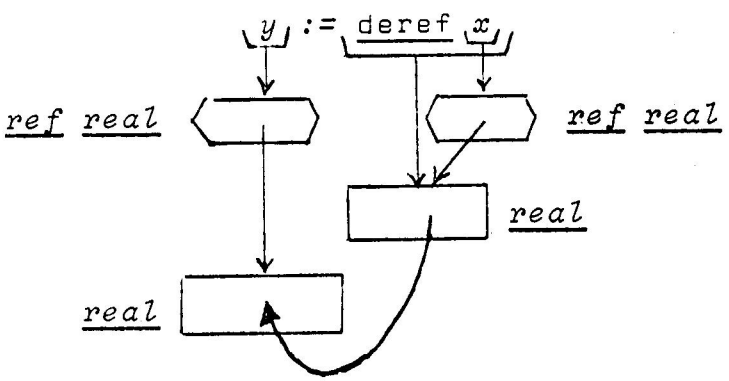


Auf der linken Seite sind wir nicht an dem *real*-Wert interessiert, den wir durch Absteigen von *ref real* erreichen würden. Was uns interessiert ist gerade die Adresse, der wir einen neuen *real*-Wert zuweisen wollen. Es findet also auf der linken Seite kein "dereferencing" statt ("soft context" vgl. 4.4.2).
Auf der rechten Seite dagegen sind wir an der Adresse nur interessiert, um den Wert zu erreichen, auf den sich die Adresse bezieht. Denn gerade diesen Wert wollen wir der Adresse der linken Seite zuweisen. Wir steigen also hier zu dem *real*-Wert ab.

Wenn wir keine impliziten "coercions" hätten, müszten wir schreiben:

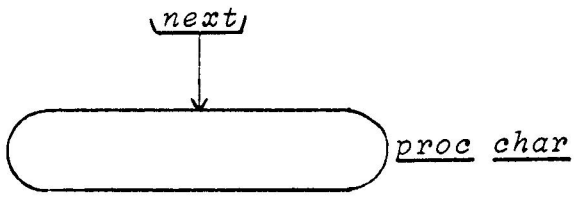
y := real(x)

Dies ist korrekt in ALGOL68 (Anwendung des cast), man braucht es aber nicht zu schreiben, weil es der Compiler für uns tut:

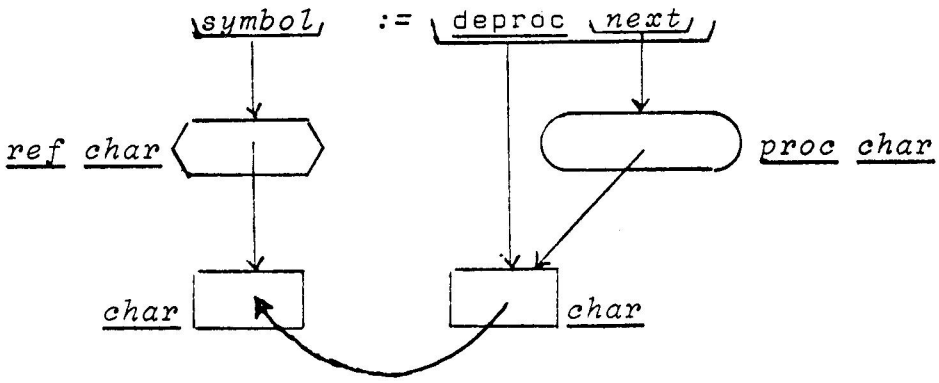


4.1.3. "deproceduring"

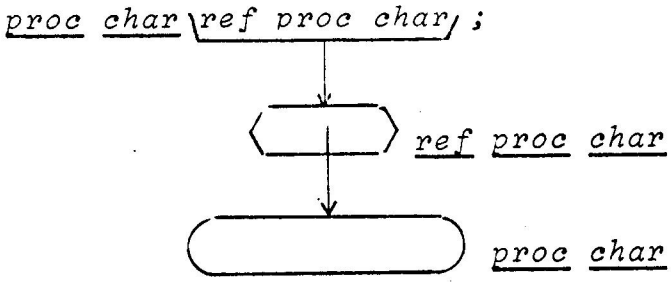
Durch das "deproceduring" wird ein *proc amode* in ein *amode* übergeführt. Diese Operation ist eigentlich trivial. Tritt ein *proc amode*-identifier (z.B. der *proc char*-identifier *next*, vgl. 3.2.2) im Programmtext auf, so hat er Zugriff zu einer Routine ohne Parameter:



In den meisten Anwendungen (jedoch nicht in allen!) will man aber den *char* haben, den diese Routine bei ihrem Aufruf als Ergebnis liefert, und nicht diese Routine selbst. Dieser Aufruf der Routine sowie die Lieferung des *char* wird durch das "deproceduring" bewirkt. So macht zum Beispiel aus *symbol := next* der Compiler:

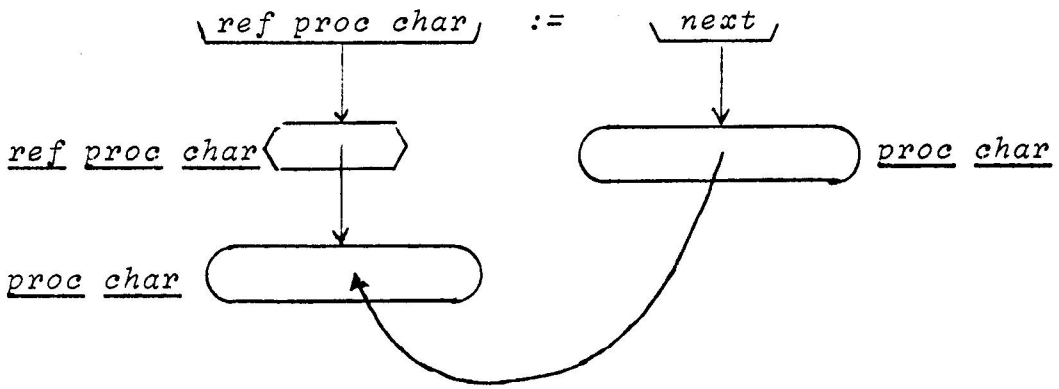


Hat man dagegen



deklariert, so hat *ref proc char* Zugriff zu der Adresse eines *proc char*, der man also ein *proc char* zuweisen kann.

Zum Beispiel:

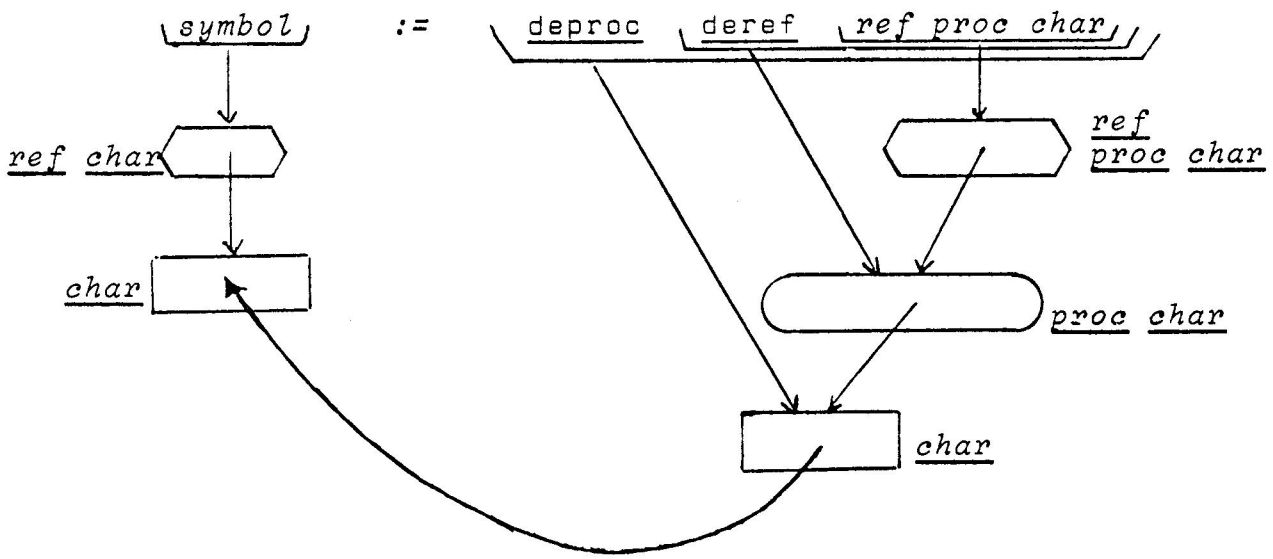


Hierbei findet auf der rechten Seite kein "deproceduring" statt, da *next* hier liefern soll, was es schon "apriori" erhalten hat, nämlich ein *proc char*.

Schreibt man jetzt aber:

symbol := ref proc char

so macht der Compiler davon:



Durch das "deproceduring" findet ein Aufruf nur von einer Routine ohne Parameter statt. Prozeduren mit Parametern werden nicht implizit durch das "deproceduring", sondern explizit durch ihren call aufgerufen. Man vergleiche die units *next* und *nextto("?")*. Bei *next* entscheidet der Kontext ob die Routine aufgerufen werden soll oder vielleicht die Routine selbst verlangt wird. Bei *nextto("?")* ist kein Zweifel möglich: das unit *nextto* erhält die Routine, *nextto("?")* ruft sie auf mit dem aktuellen Parameter *"?"*.

4.1.4. "rowing"

Auch diese Anpassung ist trivial: "rowing" überführt ein *amode* in ein []*amode*.
Beispiel:

```
string := "more than 1 character"
```

Hier is *"more than 1 character"* ein row-of-character-denotation, das man eingeführt hat, weil man sonst auf der rechten Seite

```
("m","o","r","e"," ","t","h","a","n"," ","1"," ",
 "c","h","a","r","a","c","t","e","r")
```

schreiben müszte. Man beachte jedoch, dasz:

```
"a"
```

eine character-denotation ist, und nicht ein row-of-character-denotation. Das "rowing" überführt dann hier ein *char* in ein []*char* Der Compiler übersetzt also:

```
string := "a"
```

in

```
string := row "a"
```

Auch in der assignation:

```
string := if read(symbol); "a"≤symbol and symbol≤"z"
          then symbol else "?"
            fi
```

tritt u.a. "rowing" auf:

```
string :=     if read(symbol);
                 "a"≤deref symbol and deref symbol≤"z"
            then row deref symbol else row "?"
              fi
```

Das "rowing" spielt nur eine Rolle, wenn man einem *ref flex* []*amode* (ein *amode* zuweisen will; dank des "rowing" kann man also, in "strong contexts" (wie z.B. die rechte Seite einer assignation, vgl.4.4), ein character-denotation als ein row-of character-denotation ansehen. Da ein *string* durch die mode-declaration

```
mode string = flex [1:0]char
```

standardmäszig definiert ist, so wird ein row-of-character-denotation auch als string-denotation bezeichnet.

4.1.5. "voiding"

Durch das "voiding" wird ein *amode* in ein *void* übergeführt, d.h. der *amode* wird nicht weiter verwendet.

Alle units in ALGOL68 besitzen einen Wert eines bestimmten "mode". Folgt einem unit unmittelbar ein go-on-token ";", so wird der Wert des unit beim 'Passieren' dieses go-on-token nicht weiter verwendet: man wirft diesen Wert weg, 'der Mohr hat seine Schuldigkeit getan, der Mohr kann gehen'.

Beispiel:

```
absum plusab  if read(x); num plusab 1; x>0
              then pos plusab 1; posum plusab x; +x
              elif x<0
              then neg plusab 1; negum plusab x; -x
              else num minusab 1; 0
                fi
```

Der Compiler übersetzt dies wie:

```
absum plusab  if read(x);
                  voiden(num plusab 1) ; deref x > 0
              then voiden(pos plusab 1) ;
                  voiden(posum plusab x) ; + deref x
              elif deref x < 0
              then voiden(neg plusab 1) ;
                  voiden(negum plusab 1) ; - deref x
              else voiden(num minusab 1) ; widentoreal 0
                fi
```

Durch das "voiding" sieht man klar und deutlich, welchen Wert eine serial-clause liefert: den Wert ihres letzten unit, die Werte aller anderen units werden weggeworfen.

Im obigen Beispiel beachte man noch, dasz das erste unit der enquiry-clause *read(x)* schon ein *void* erhält (der Compiler braucht es also nicht mehr zu "voiden"), da doch die "routine" der Standard-Prozedur *read* bei ihrem Aufruf ein *void* liefert.

In 2.8.2 haben wir gesehen, dasz der Wert einer closed-collateral-clause das Gesamtgebilde der Werte ihrer units ist; dieses Gesamtgebilde ist dabei stets entweder ein "multiple value" oder ein "structured value" (man vergleiche 2.8.2 die Beispiele a -- e). Man vergleiche auch die collateral-clause nach dem label *result:* in Beispiel {7} in 1.4.6. Ob dann eine collateral-clause ein "multiple-value" oder ein "structured-value" ist, entscheidet der ("strong" vgl. 4.4.2) Kontext.

Eine collateral-clause kann aber auch als phrase (als unit) in einer serial-clause auftreten. Tritt sie dann dort nicht als letztes unit auf, so kann sie als statement angesehen werden, dies bedeutet: die Werte aller ihrer units werden weggeworfen.

Eine collateral-clause als statement begegnen wir in Beispiel {2} (vgl. 1.4.2). Der Compiler übersetzt:

```
{2}     begin   int  k, n;
                real next, sum, squm;

                read(n) ;

                voiden ( sum:= deref (sum:= widentoreal 0) ) ;

                voiden ( k:=1 ) ;

        do:     read(next) ;

                ( voiden ( sum:= deref  sum + deref next )      ,
                  voiden (squm:= deref squm + deref next ↑ 2 ) ,
                  voiden (    k:= deref k     +     1        )
                ) ;

                if deref k ≤ deref n then goto do fi ;

                print ( ( deref sum / deref n ,
                          sqrt ( deref n × deref squm
                                 - deref sum ↑ 2 ) / deref n
                        )
                      )

        end
```

4.1.6. Jump und *skip*

Bisher haben wir den Wert, den ein jump liefert, als *void* betrachtet. Offiziell (also nach dem Report) stimmt dies nicht ganz.
Formal-syntaktisch kann ein jump einen Wert eines jeden, vom Kontext geforderten "mode" liefern.

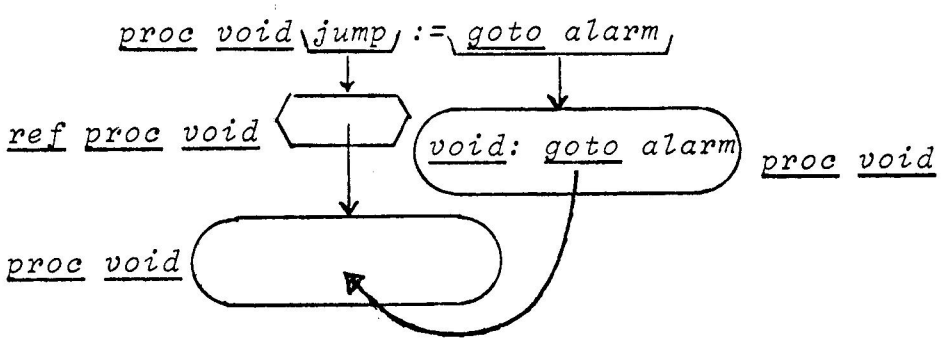
In der folgenden assignation

y := if x≥0 then sqrt(x) else goto alarm fi

wird der jump *goto alarm* vom Compiler so behandelt, als ob er einen *real* liefern würde, damit die gesamte assignation formal-syntaktisch korrekt ist.
Niemand (auch nicht der Compiler) wird sich natürlich hierbei um diesen *real*-Wert kümmern; denn zur "runtime", wenn die obige assignation ausgeführt wird, wird in dem Fall, dasz *x* einen negativen Wert liefert, stets ein Sprung zu der durch den label-identifier *alarm* markierten Stelle im Programmtext ausgeführt.
Betrachtet man also die leere Menge (*void*) als Teilmenge aller Mengen, so erscheint es berechtigt zu sagen, ein jump liefert einen *void*. Für uns bedeutet *void* immer: ein Wegwerfwert.

Allerdings, wenn der Kontext eine "routine" verlangt, so liefert der jump eine "routine", die aus dem jump allein besteht.

Beispiel:



Die "elaboration" von *goto alarm* liefert in diesem Fall keinen Sprung, sondern eine "routine" (eine *proc void*), die der *proc void*-Variablen *jump* zugewiesen wird.

Erst wenn die Variable *jump* als applied-identifier in einem Kontext auftritt, wo weder ein *ref proc void* noch ein *proc void* verlangt wird, wird bei ihrer "elaboration" die ihr zugewiesene

Routine aufgerufen. Dies bedeutet, dasz dann erst an dieser Stelle (nach dem "dereferencing" und "deproceduring") ein Sprung ausgeführt wird.

Es gibt neben dem jump noch ein weiteres unit, welches ebenfalls einen Wert eines jeden, vom Kontext geforderten "mode" liefert. Es ist dies der *skip*. Das unit *skip* kann man dann schreiben, wenn die Syntax ein unit fordert, man jedoch nicht weisz, welches unit man schreiben soll.
Der *skip* also ist das ALGOL68-'dummy statement' (man vergleiche Beispiel {5} in 1.4.4). In den meisten Fällen kann man aber wegen der Nachgiebigkeit der Syntax den *skip* vermeiden.

Beispiele:

```
if x≥0 then y:=sqrt(x) else skip fi
```

Hier wird ein jeder wohl schreiben:

```
if x≥0 then y:=sqrt(x) fi
```

```
case i
  in unit1 , unit2 , unit3 ,  unit4
 out skip
esac
```

Hier schreibt man natürlich :

```
case i
  in unit1 , unit2 , unit3 , unit4
esac
```

Man vergleiche auch wie in Beispiel {5'} in 1.4.6 der *skip* von Beispiel {5} verschwunden ist.

So man will, kann man den *skip* als einen jump unmittelbar hinter das (in der textlichen Reihenfolge) nächste go-on-token auffassen.

Beispiel:

```
                    goto
case i in unit1 , skip , unit3 esac ; unit ;

( serial-clause ; exit: skip ) ; unit ;
                        goto
```

4.2. Verwandte "modes"

4.2.1. *bool* und *bits*

Aus dem Grund-"mode" *bool* (vgl. 1.1.2) wird durch die standardmäszige declaration

mode bits = struct ((1: bits width)bool ℵ)

der "mode" *bits* definiert.

Mit Hilfe von *bits* (ebenso wie mit *bytes*, vgl. 4.2.2) hat der Programmierer Zugriff zu einzelnen Maschinenworten im Speicher. Denn ein *bits* ist kein "multiple value", sondern eine Folge von *bool*s fest vorgegebener Länge, aufgefaszt als ein Maschinenwort. Zu diesem Maschinenwort wird nicht über einen etwaigen "descriptor" (wie bei einem "multiple value"), sondern direkt zugegriffen, unter Ausnutzung aller hierfür vorhandenen "hardware"-Möglichkeiten der Maschine, auf der die Sprache implementiert ist. Um dem Programmierer diese Möglichkeit des effizienten Zugriffs zu geben und ihn daran zu hindern, ein *bits* als ein "multiple value" zu behandeln, so ist diese Folge von *bool*s in eine Struktur eingebettet, deren field-selector ℵ von dem Programmierer nicht benutzt werden darf.
Durch ein *bits* wird ein Maschinenwort als eine Folge von 'bits' (d.h. *bool*s) angesehen. Dabei bestimmt *bits width* die Anzahl der 'bits' in einem Maschinenwort, gibt also die Wortlänge an. Die Konstante *bits width* ist implementationsabhängig und standardmäszig definiert.
Für *bits* sind Operatoren definiert, die die gebräuchlichen Maschinenoperationen durchführen (vgl. auch 5.2.6). Insbesondere kann man mit Hilfe des Operators *elem* ein einzelnes *bool* aus einem *bits* auswählen.

Zunächst ein Überblick der Standard-Operatoren für *bits*; man vergleiche auch 5.2.6.

operator	priority	left operand	right operand	result
abs	*10*		*bits*	*int*
bin	*10*		*int*	*bits*
elem	*7*	*int*	*bits*	*bool*
and *or*	*3* *2*	*bits*	*bits*	*bits*
↑ ×× *up*	*8*	*bits*	*int*	*bits*
= *eq* ≠ *ne* ≤ <= *le* ≥ >= *ge*	*4* *4* *5* *5*	*bits*	*bits*	*bool*

Für *bits* gibtes (ebenso wie für die Elemente von Grund-"modes") eigene denotations. Diese werden in der "radix"-Form angegeben; man hat also die Möglichkeit, ein *bits* in demjenigen Zahlsystem zu bezeichnen, in welchem bei einer bestimmten Maschine die einzelnen Maschinenworte abgespeichert werden. Möglich sind hierbei die Zahlsysteme zur Basis ("radix") 2, 4, 8 oder 16.
So bezeichnen zum Beispiel die bits-denotations:

2r111101 , *4r331* , *8r75* und *16r3d*

jeweils das gleiche *bits*, je nach verwendetem Zahlsystem. Hierbei gilt: links von dem *r* steht die Basis des Zahlsystems, rechts davon die entsprechende Darstellung des *bits*. Die Buchstaben *a*,*b*,*c*, *d*,*e*,*f* werden in der "radix"-Form *16r* für die 'Ziffern' *10*,*11*,*12*, *13*,*14*,*15* benutzt.
Ist z.B. *bits width* = *32* (hat also ein Maschinenwort die Länge 32), so werden die Werte der oben angegebenen bits-denotations stets durch die gleiche Folge von 'bits' im Speicher dargestelt wie der Wert von

2r00000000000000000000000000111101

Innerhalb des Maschinenwortes werden die dem bits-denotation entsprechenden 'bits' rechtsbündig abgespeichert und nach links mit *false* (also *0*) aufgefühlt.

4.2.2. *char* und *bytes*

Neben dem *string* wird aus dem Grund-"mode" *char* ein weiterer "mode" standardmäszig definiert:

mode bytes = struct ((1: bytes width)char ℵ)

Wie ein *bits*, so ist ach ein *bytes* kein "multiple value", sondern eine Folge von *chars* fest vorgegebener Länge, aufgefaszt als ein Maschinenwort. Hierbei wird ein Maschinenwort als eine Folge von 'bytes' betrachtet, wobei ein 'byte' diejenige Folge von *bools* ('bits') angibt, die den Wert eines *char* darstellt. Der Zugriff innerhalb des Speichers zu einem *bytes* geschieht auch hier nicht über einen etwaigen "descriptor", sondern direkt, unter Ausnutzung der hierfür vorhandenen "hardware"-Möglichkeiten der Maschine. Deswegen ist auch hier die Folge von *chars* (die ein *bytes* bilden) in eine Struktur eingebettet, deren field-selector ℵ von dem Programmierer nicht benutzt werden kann. Ein *bytes* unterscheidet sich also von einem *string* (welches eine flexible Reihe von *chars* ist) darin, dasz der Zugriff zu einem *bytes* in den meisten Implementationen sehr viel effizienter erfolgen kann als der Zugriff zu einem *string*.
Die Anzahl der 'bytes' innerhalb eines Maschinenwortes wird durch *bytes width* bestimmt; ebenso wie *bits width* ist hierbei auch *bytes width* implementationsabhängig und standardmäszig definiert.

Wie für *bits*, sind auch für *bytes* Operatoren definiert, die die gebräuchlichen 'Teilwort'-Maschinenoperationen effizient durchführen:

operator	priority	left operand	right operand	result
elem	7	*int*	*bytes*	*char*
ctb	10		*string*	*bytes*
= *eq*	4	*bytes*	*bytes*	*bool*
≠ /= *ne*	4			
< *lt*	5			
> *gt*	5			
≤ <= *le*	5			
≥ >= *ge*	5			

4.2.3. *long* und *short* "modes"

Bei der Implementation werden für alle *int*s einen gleichgroszen Speicherplatz zur Verfügung gestellt, in welchem die *int*s intern dargestellt werden. Das gleiche wird für *real*s, *compl*s, *bits* und *bytes* gelten, wobei dann natürlich z.B. die Speicherplätze für die *ints* und die Speicherplätze für die *real*s unterschiedliche Grösze haben können (dies wird bestimmt der Fall sein für *real*s und *compl*s).

Man hat nun in ALGOL68 die Möglichkeit, die einzelnen Werte jeder der oben genannten Mengen in verschiedenen groszen Speicherplätzen intern unterzubringen.
Dies kann für den Programmierer sinnvoll sein, wenn er z.B. eine gröszere Genauigkeit seiner intern dargestellten Werte haben möchte oder aber ihm der üblicherweise für seine Werte zur Verfügung gestellte Speicherraum zu grosz ist, er z.B. zwischen einer gröszeren Anzahl von Werten des gleichen "mode" unterscheiden möchte; ganz allgemein: wenn der Programmierer mehr oder weniger Information als üblich in seine Werte stecken möchte.

Mit Hilfe der 'mode-maker' *long* und *short* kann der Programmierer dies für *int*s, *real*s, *compl*s, *bits* und *bytes* tun.
Durch mehrmaliges voraussetzen entweder von *long* oder von *short* an den "mode" (oder denotation) eines Wertes dieser Mengen werden die jeweiligen Werte zu Werten unterschiedlicher Länge zusammengefaszt.

Deklariert man zum Beispiel:

long long int mmmm; long int mmm; int mm; short int m;

so beziehen sich die Adressen von *mmmm*, *mmm*, *mm* und *m* jeweils auf *int*-Werte der Länge 2, 1, 0 und -1.

Natürlich kann man dann davon ausgehen, dasz i.a. für den *int*-Wert der Länge 2, bzw.1, mehr (vielleicht dreimal, bzw. zweimal soviel) und für den *int*-Wert der Länge -1 weniger (vielleicht halb soviel) Speicherplatz zur Verfügung steht als für den *int*-Wert der Länge 0. Dies ist jedoch sehr abhängig von der Implementations-Maschine.

Die maximale Werte verschiedener Länge sind standardmäszig gegeben durch die *int*-Konstanten

max long int , *max long long int* , usw.
und *max long real* , *small long real* , *max long long real* , usw.

Nun wird im allgemeinen eine gegebene Implementation nicht unter Werten beliebiger Länge unterscheiden, vielmehr werden ab einer bestimmten Länge (sowohl nach oben als auch nach unten) die Werte von gleicher Genauigkeit angesehen (es könnte sogar so sein dasz eine Implementation nur Werte der Länge 0 zur Verfügung stellt und vielleicht nur am Sonntag ein wenig mehr).
Zwischen wieviel verschiedenen Längen eines *int* bzw. *real* (und damit auch eines *compl*) jeweils unterschieden wird, wird standarmäszig durch die *int*-Konstanten

int lengths und *int shorts*
bzw. *real lengths* und *real shorts*

angegeben.
Anhand dieser *int*-Konstanten kann der Programmierer feststellen, wieviel *long*s bzw. *short*s er benutzen kann, um Werte unterschiedlicher Länge zu erhalten.

Ist etwa bei einer Implementation

int lengths = 3 und *int shorts* = 1

so wird bei dieser Implementation nur zwischen fünf *int*-Werte verschiedener Länge unterschieden, nämlich

long long long int , *long long int* , *long int* , *int*
und *short int*

Deklariert dann der Programmierer in seinem Programm etwa

long long long long long long long long int mmmmmmmmmm;
und *short short int n;*

so sind allerdings die "modes" von *mmmmmmmmmm* und *n* verschieden von allen andere *long* --- *long int*- bzw. *short* --- *short int*-Werte, es werden aber die Werte dieser (zu lange bzw. zu kurze) "modes" abgespeichert als wären diese *long long long int*s bzw. *short int*s.

Wir haben in 4.2.1 und 4.2.2 gesehen, dasz sie Anzahl der 'bits' (*bool*s) in einem *bits* durch *bits width* und die Anzahl der 'bytes' (*char*s) durch *bytes width* bestimmt ist. Für die einzelnen *bits* bzw. *bytes* unterschiedlicher Länge (wieviele davon unterschieden werden, wird wieder standardmäszig durch die *int*-Konstanten *bits lengths* und *bits shorts* bzw. *bytes lengths* und *bytes shorts* bestimmt) wird dann die Anzahl der in ihnen enthaltenen 'bits'

bzw. 'bytes' jeweils durch (dieser Paragraph ist longlonglong-langweilig)

bits width , *long bits width* , *long long bits width* , etc
short bits width , *short short bits width* , etc.

bzw. durch

bytes width , *long bytes width* , *long long bytes width* , etc.
short bytes width , *short short bytes width* , etc.

bestimmt.

Wie wir in 4.2.4 sehen werden, gibt es kein "widening" von z.B. einem *real* der Länge 1 zu einem *real* der Länge 2.
Hierfür haben wir jedoch den monadischen Operatoren *leng* und *shorten*:

leng	'erweitert' seinen (rechten) Operand zu einem Wert (*int*, *real*, *compl*, *bits* oder *bytes*) mit einem *long* mehr oder einem *short* weniger.
shorten	'verengt' seinen (rechten) Operand zu einem Wert (*int*, *real*, *compl*, *bits* oder *bytes*) mit einem *short* mehr oder einem *long* weniger.

Die ('mode-maker') symbols *long* und *short* braucht man auch in denotations für Werte in *long*- bzw. *short*-"modes".

Man schreibt also:

long real pii = long 3.14159265358968323756264338

Wenn man das *long* hier wegläszt, macht man einen syntaktischen Fehler (es gibt kein "widening" von *real* nach *long real*, vgl. 4.2.4). Sogar musz man auch schreiben zum Beispiel:

long int zero = long 0
und *short int o = short 0*

Und ebenso bei *bytes* und *bits*:

mode byte = short bits
byte mask = short 2r11111111

Alle Operatoren die mit *int*, *real*, *compl*, *bits* und *bytes* standardmäszig definiert worden sind, sind auch (und genau so) definiert für den entsprechenden *long*- und *short*-"modes". Man beachte jedoch, dasz es auch dabei kein "widening" oder "narrowing" gibt zwischen *longs* oder *shorts*.
Hat man also ein "x" für links *int* oder *real* oder *compl* und rechts *int* oder *real* oder *compl*; dann hat man deswegen auch ein "x" für links *long int* (*long real*, *long compl*) und rechts *long int* (*long real*, *long compl*), nicht aber für z.B. links *int* (*real*, *compl*) und rechts(*long int*, *long real*, *long compl*) und so weiter.

Zum Schlusz noch ein Beispiel:

```
proc inprod  =  co a procedure that computes the innerproduct
                   of two rows-of-real, which are possibly
                   unequal in length and (or) index-range.
                   both rows will be considered to consist
                   of zeros where they do not cope with the
                   index-range of the other row
                co

                ( ref []real u, v) real:

     (  int lwbu = lwb u , lwbv = lwb v ,
            upbu = upb u , upbv = upb v ;

        long real sum := long 0.0 ;

         for i
        from if lwbv>lwbu then lwbv else lwbu fi
          to if ipbv<upbu then upbv else upbu fi
          do
             sum plusab leng u[i] × leng v[i]
          od ;

        shorten sum

     )
```

4.2.4. "widening"

Die Anpassungsoperation "widening" überführt ein *int* in ein *real* und ein *real* in ein *compl* (jeweils unter Beibehaltung der Länge) sowie ein *bits* in ein ()*bool* und ein *bytes* in ein ()*char*.

Der Compiler wird das "widening" nur im sogenannten "strong context" (vgl. 4.4.2) vornehmen, so z.B. auf der rechten Seite einer assignation oder auf der rechten Seite einer identifier-declaration; jedoch z.B. nicht in einer formula. Wir werden später darauf zurückkommen (vgl. 5.1.3).

Wenn der Kontext ein *real* fordert, der "coercend" apriori ein *int* erhält, so kann dieses *int* erweitert werden zu einem *real* gleicher Länge: dies bewirkt die "coercion" (implizite Operation) widentoreal. Durch die "coercion" widentocompl wird ein *real* erweitert zu einem *compl* gleicher Länge.

Beispiele:

x := 37 , *z := 37.0* ,
z := 37

Der Compiler übersetzt:

x := widentoreal *37* , *z :=* widentocompl *37.0* ,
z := widentocompl widentoreal *37*

In ALGOL68 gibt es kein "narrowing" (Einengen): man kann in vielen Fällen ("strong context") von *int* zu *real* und weiter zu *compl* mit Hilfe des impliziten "transfer" übergehen, jedoch nicht umgekehrt (wie es z.B. in ALGOL60 möglich ist).
Man musz in solchen Fällen schreiben:

i := *round* *x* oder *i :=* *entier* *x* ,

wobei *round* und *entier* monadische Standard-Operatoren sind (vgl. 1.3.4).
Die Ursache, warum ALGOL68 kein "narrowing" kennt, liegt daran, dasz zwar der Übergang *int* →*real*→*compl* mathematisch eindeutig ist, jedoch nicht die umgekehrte Richtung (der eine will *round*, der andere *entier*, und für *compl*→*real* kann man sogar wählen zwischen *re*, *im* und *abs*).

Auch für die "hardware"-"modes" *bits* und *bytes* ist in ALGOL68 (im "strong context") ein "widening" definiert: ein *bits* bzw. ein *bytes* kann in ein "multiple value" und zwar in ein ()*bool* bzw.()*char* übergeführt werden. Dabei bestehen dann diese beiden "multiple values" aus den in dem *bits* bzw. *bytes* enthaltenen *bools* bzw. *chars*, und die "descriptors" sind (*1: bits width*) bzw. (*1: bytes width*).

Beispiel:

Hat man deklariert:

```
string text; bytes charword := bytespack("1968");
(1: bitswidth)bool boolrow;
```

und schreibt dann:

```
boolrow  := 2r111101 ;   text := charword ;
```

so macht der Compiler daraus:

```
boolrow  := widentorowofbool 2r111101 ;
    text := widentorowofchar deref charword ;
```

Hiernach bezieht sich die Adresse von *boolrow* auf das "multiple value", welches aus den *bools* von *2r111101* besteht, während sich die Adresse von *text* auf das "multiple value" von *"1968"* bezieht.

Die Prozedur *bytespack* ist standardmäszig definiert und wandelt ein *string* in ein *bytes* um, wobei die einzelnen Zeichen innerhalb des *string* als 'bytes' dargestellt werden.

Das "widening" geschieht nur für die "transfers":

int → *real* , *long int* → *long real* , usw.

real → *compl* , *long real* → *long compl* , usw.

bits → (*1: bits width*)*bool* ,

long bits → (*1: long bits width*)*bool* , usw.

bytes → (*1: bytes width*)*char* ,

long bytes → (*1: long bytes width*)*char* , usw.

Für den Übergang zwischen *int*, *real*, *compl*, *bits* und *bytes* verschiedener Länge, verwende man die Operatoren *leng* und *shorten* (vgl. 4.2.3).

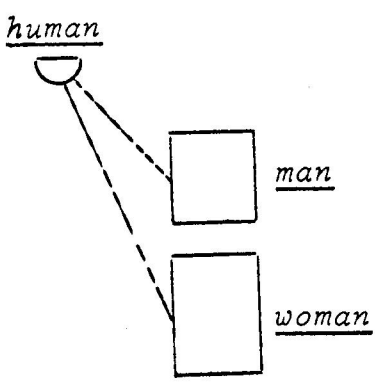
4.3. "United-modes"

4.3.1. Der Begriff *union*

Für viele Anwendungen ist es oft nützlich, auch einen "mode" für die Vereinigung von verschiedenen Werte-Mengen zu haben. Die Vereinigung verschiedener solcher Mengen wird zustande gebracht durch den 'mode-maker' *union(,)*:

mode human = union(man,woman)

Man kann nun von einem *human* sprechen und damit ein Element aus der Vereinigung aller Werte vom "mode" *man* und *woman* meinen. Ein *human* ist dann also stets entweder ein *man* oder ein *woman*. Einen neuen Wert vom "mode" *human* gibt es also nicht. Wir stellen ein *human* im Bild folgendermaszen dar:



In diesem Bild kennzeichnet *human* eine Art "descriptor", der zur 'runtime' (!) angibt, welcher "mode" für *human* zutrifft, ob der *human* also ein *man* oder ein *woman* ist.

"United-modes" sind kommutativ, assoziativ und 'absorbierend':

mode namuh = union(woman,man)

spezifiziert den gleichen "mode" wie *human*. Auch:

mode birc = union(bool,int,real,char) ,
mode bbirc = union(bool,bool,int,real,char) ,
mode cribb = union(char,real,union(int,bool),bool)

spezifizieren alle den gleichen "mode".

Mit einem *union* dürfen allerdings nicht solche Werte zu einem *union* vereinigt werden, deren "modes" sich durch voraussetzen der 'mode-maker' *ref* und *proc* an *amode* ergeben.
Diese so (durch "deproceduring" bzw. "dereferencing") auseinander hervorgehenden "modes" sind zu sehr miteinander verbunden, so dasz in manchem Kontext es nicht eindeutig ist, in welcher Reihenfolge die drei "coercions" deref bzw. deproc sowie unite (vgl. 4.3.3) anzuwenden sind.

So sind zum Beispiel (neben den in 3.1.6 genannten) die folgende mode-declarations nicht erlaubt:

mode refrelated = union(ref amode, amode)
mode procrelated = union(proc amode, amode)

mode refprocrelated = union(ref proc ref amode, ref amode)

Dagegen ist aber zum Beispiel:

mode notrelated = union(ref proc ref amode, proc amode)

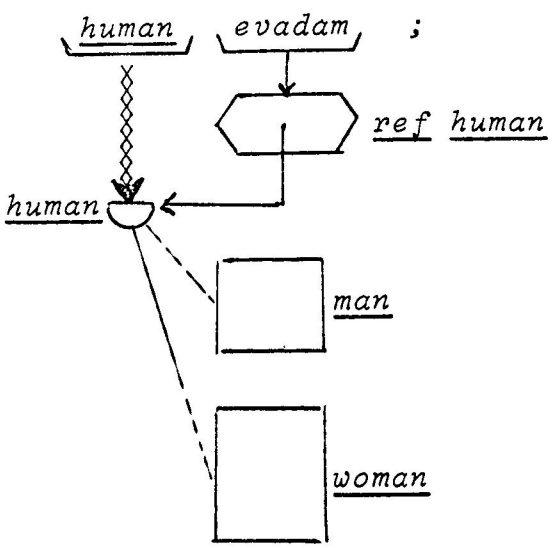
syntaktisch einwandfrei, da man aus *ref proc ref amode* durch deref und deproc niemals auf *proc amode* kommen kann.

4.3.2. Die *union*-Variable

Betrachten wir die folgende variable-declaration:

human evadam ;

Wir erhalten dann folgendes Bild:



Auf dem "runtime-stack" wird hierbei hinreichend Speicherraum reserviert sowohl für ein *man* als auch für ein *woman*.
Der *ref human* identifier *evadam* hat Zugriff zu dieser Adresse.
Überdies wird innerhalb des internen Objektes des *human* Speicherraum reserviert für die Administration des geltenden "mode".

Man hat allerdings die Möglichkeit, mit Hilfe der conformity-clause (vgl. 4.3.4) festzustellen, welcher Wert von welchem "mode" zu einem bestimmten Zeitpunkt der "elaboration" tatsächlich vorliegt.

4.3.3. "Uniting" und Zuweisung an eine *union*-Variable

Bei allen bisher betrachteten "coercions" werden stets zur "runtime" gewisse (Grund-)Aktionen ausgeführt, um einen Wert eines bestimmten "mode" in einen Wert eines anderen "mode" zu überführen. Anschaulich gesprochen bedeutet dies: ein Wert eines bestimmten "mode" wird aus der Klasse aller Werte dieses "mode" (mit mehr oder weniger Gewalt) in eine andere Klasse von Werten übergeführt.
Beim "uniting" wird dagegen der "mode" eines zur "runtime" bestimmten Wertes nicht geändert. Solch einem Wert wird aber durch das "uniting" insofern mehr Spielraum gelassen, als dasz er nicht mehr als Element einer einzigen Klasse von Werten betrachtet wird, sondern vielmehr als Element der Vereinigung mehrerer (endlich vieler) Klassen.
Formal syntaktisch kann man jedoch sagen, dasz ein "mode" *amode* in den "mode" *union(amode,)* übergeführt wird.

Sei zum Beispiel:

mode number = union(int,real,compl)

Deklarieren wir dann

number irc; proc f = (number u) number: unit ;

und betrachten die nachfolgende assignation

irc := f(3.14) ;

Bei der "elaboration" dieses procedure-call (vgl. 3.2.4) auf der rechten Seite begegnen wir der folgenden identifier-declaration (durch die die Parameterübergabe vorgenommen wird)

number u = 3.14

Hierdurch soll also *u* den Wert von *3.14*, also ein *real* erhalten. Dies ist jedoch erst dann möglich, nachdem der Compiler durch das "uniting" festgestellt hat, dasz dieser *real* (den *3.14* erhält) ein Element der durch *number* bestimmten Vereinigung aller *int*s, *real*s und *compl*s ist.

Der Compiler macht also aus obiger identifier-declaration

number u = unite 3.14

Dadurch, dasz durch das "uniting" ein Wert als ein Element einer Vereinigung mehrerer Klassen von Werten aufgefaszt wird, hat man z.B. die Möglichkeit, der Adresse einer einzigen Variablen Werte von verschiedenen "modes" zuzuweisen.
Betrachten wir dazu z.B. die folgenden variable-declarations

number *irc* ;
int *i*, *real* *r*, *compl* *c*;

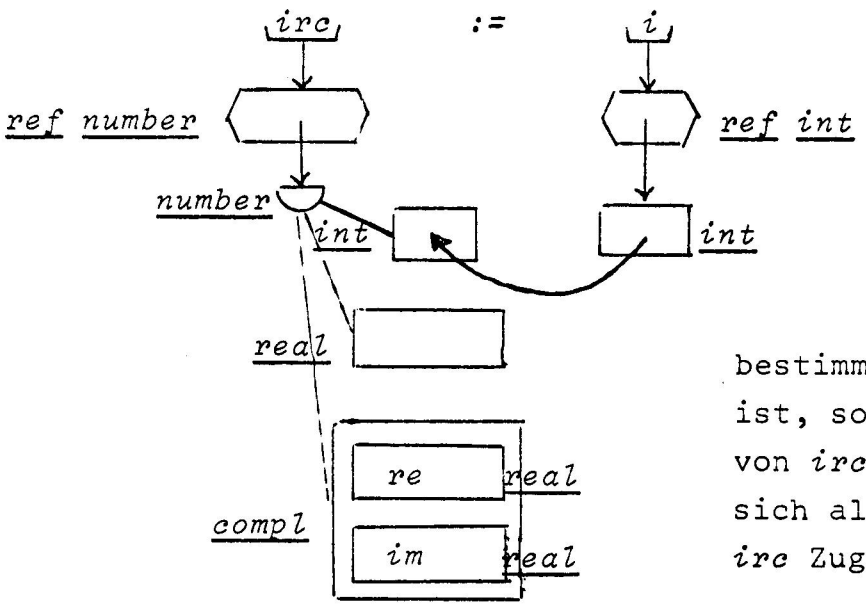
sowie die nachfolgende assignation

irc := *i*

Der Compiler macht daraus

irc := unite deref *i*

Die "elaboration" dieser assignation ergibt folgendes:



Nachdem durch das "uniting" festgestellt wurde, dasz der *int*, den *i* nach dem "dereferencing" liefert, ein Element der durch *number* bestimmten *union*(*int*,*real*,*compl*) ist, so wird dieser *int* der Adresse von *irc* zugewiesen. Danach bezieht sich also der *ref* *number* zu dem *irc* Zugriff hat, auf diesen *int*.

Entsprechende Bilder ergeben sich bei:

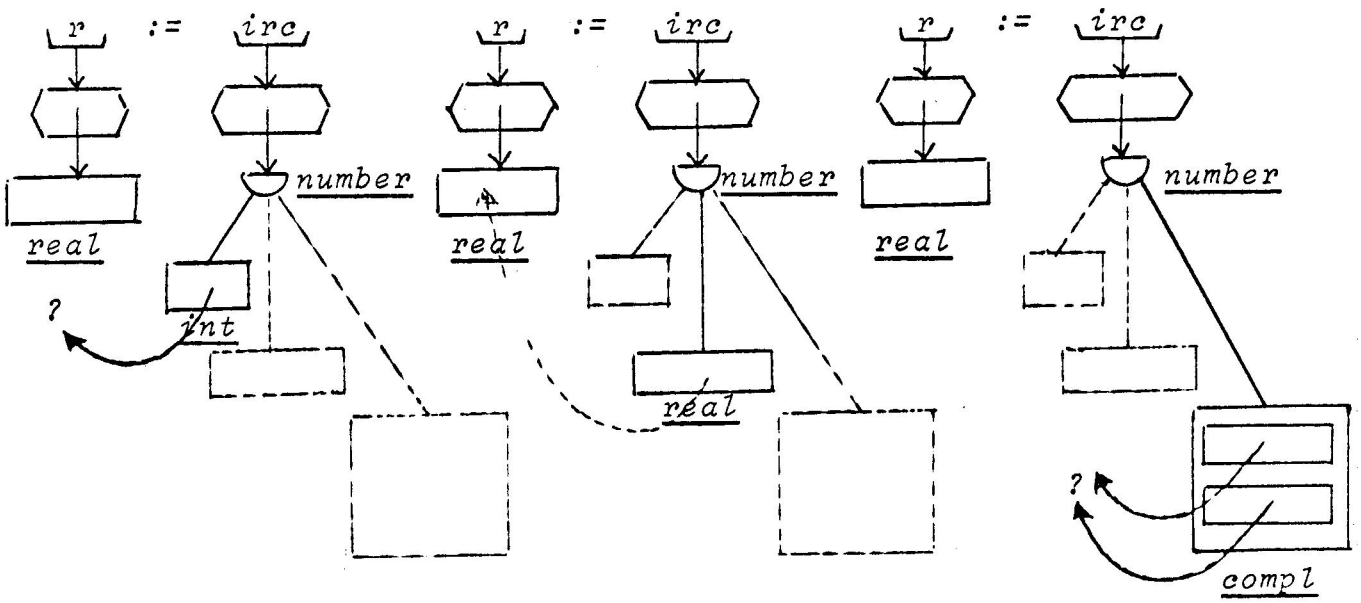
irc := *r* also *irc* := unite deref *r*
irc := *c* also *irc* := unite deref *c*

Durch den 'mode-descriptor' in *number* wird festgestellt, welcher Wert nach jeder assignation an dem für *number* reservierten Speicherplatz abgespeichert wurde. In 4.3.4 werden wir lernen, wie man durch eine conformity-clause den geltenden "mode" eines *union* abfragen kann.

Im Gegensatz zu den letzten drei betrachteten assignations ist die 'umgekehrte' Zuweisung, wie zum Beispiel:

r := irc

syntaktisch inkorrekt.
Dadurch sollen mögliche unsinnige Zuweisungen schon zur "compile-time" verhindert werden. Täte man dies nicht, so würde man die "elaboration" der assignation mit einer aufwendigen "runtime"-Prüfung überladen. Denn erst zur "runtime" kann festgestellt werden, welchen "mode" der *number*, den *irc* liefert, tatsächlich hat. Und erst dann könnte man entscheiden, ob dieser Wert zugewiesen werden kann oder nicht. In unserem Beispiel wäre eine Zuweisung nur möglich wenn der *number* ein *real* ist:



Um also "runtime"-Prüfungen nicht der assignation aufzuladen, so verbietet man von vornherein diese Form der Zuweisung.

4.3.4. Die conformity-clause

Wie mit Hilfe der case-clause (vgl. 2.7), so kann man auch mit der conformity-clause innerhalb eines Programms unter mehreren möglichen Programmteilen einen auswählen, in welchem dann die "elaboration" fortgesetzt wird. Bei einer conformity-clause hängt diese Auswahl jedoch nicht von einem bestimmten *int*-Wert, sondern von dem "mode" eines *union* ab.
Die syntaktische Struktur der conformity-clause haben wir bereits schon in 2.6.1 gesehen. In der conformity-clause werden die einzelnen units in der in-choice-clause zu einer dem "mode"-Vergleich entsprechenden Form erweitert: diese so erweiterten units können dann gewissermaszen als routine-texts angesehen werden.

Beispiel:

```
mode strumber = union(string,number)
```

Ein *strumber* ist also ein *union(string,int,real,compl)*

```
strumber cris ;
string string, int int, real real, compl compl
```

Folgende sind ohne weiteres möglich:

cris := "1968"	und auch	*cris := string*
cris := 1968	und auch	*cris := int*
cris := 1968.0	und auch	*cris := real*
cris := (1900.0 , 68.0)	und auch	*cris := compl*

Nicht gestattet sind *string:=cris* , *int:=cris* , *real:=cris* und *compl:=cris* (vgl. 4.3.3).

Will man nun doch, in Abhängigkeit von dem "mode" des *strumber*, auf den sich die Adresse von *cris* bezieht, eine dieser Zuweisungen vornehmen, so kann man z.B. schreiben:

```
int mode := case cris
              in (string s): (string:=s; 0) ,
                 (    int i): (    int:=i; 1) ,
                 (   real r): (   real:=r; 2) ,
                 ( compl c): ( compl:=c; 3)
            esac
```

Der *int*-Wert auf den sich die Adresse von *mode* bezieht, gibt dann an, welche der vier mögliche assignations stattgefunden hat.

In einer conformity-clause musz die enquiry-clause (im obigen Beispiel besteht sie nur aus dem identifier *cris*) stets einen *union* liefern. Vor den einzelnen units innerhalb der in-choice-clause stehen jeweils sogenannte specifications (im Beispiel *(string s):*, *(int i):*, *(real r):* und *(compl c):*), die die Auswahl der verschiedenen units ermöglichen:

Die declarer in den specifications spezifizieren jeweils einen bestimmten "mode". Jeder dieser "modes" wird mit dem "mode" des *union* der enquiry-clause verglichen. Stimmt der "mode" dieses *union* mit einem dieser "modes" überein oder kann er durch das "uniting" in einen dieser "modes" übergeführt werden, so wird das der entsprechenden specification folgende unit ausgewählt und mit ihm die "elaboration" des Programms fortgesetzt. Wird dagegen keine Übereinstimmung der "modes" festgestellt, so wird die "elaboration" mit der out-choice-clause fortgesetzt.

Eine specification kann (braucht jedoch nicht) einen identifier enthalten (im Beispiel etwa *i* in *(int i):*). Dieser identifier wird durch seine specification definiert, und er kann dann innerhalb des nachfolgenden unit (aber nur dort) auftreten. Bei der Auswahl erhält dann der entsprechende identifier (wenn vorhanden) den Wert des *union* der enquiry-clause. Der Programmierer kann diesen Wert selbstverständlich in dem nachfolgenden unit verwenden; hat er dagegen kein Interesse an dem Wert (also nur an dem "mode"), so kann er sich in der specification auf den declarer beschränken.

Die einzelnen units in der conformity-clause kann man zusammen mit ihren specifications als eine Art routine-text auffassen. Dabei hat solch ein routine-text genau einen 'formalen Parameter', der auch aus dem declarer allein bestehen kann.

Im übrigen ist die syntaktische Struktur (inklusive aller in 2.7.1 gegebenen Abkürzungen, Schachtelungen und alternativen Schreibweisen) der conformity-clause gleich der der case-clause.

4.4. Die Durchschlagskraft des Kontextes

4.4.1. Der Begriff der syntaktische Position

Unter der syntaktische Position eines unit versteht man diejenige Stelle innerhalb des syntaktischen Kontextes, an der dieses unit auftritt. Beispiel :

(1:n)real row1, row2; int h, i, k; real x, y; fun f; man henry8; (1:n)(1:n)real vecrow;

Wir betrachten nun:

a)	*row1 := row2*	h)	*real(i)*
b)	*row2 := row1*	i)	*f := sin*
c)	*(1:n)real row = row1*	j)	*f := (real a)real: sin(a)/a*
d)	*read(row1)*	k)	*y := f(x)*
e)	*row1 + row2*	l)	*wife of henry8*
f)	*row1(h:k)*	m)	*vecrow(h:k)(i)*
g)	*y := row1(i)*	n)	*row1(i)+vecrow(h)(k)*

In a) ist die syntaktische Position von *row1* die einer destination, in b) dagegen die einer source. In c) ebenfalls die einer source, und in d) ist *row1* ein actual-parameter sowie *x* in k). In e) ist *row1* ein Operand und in f), g) und n) steht *row1* in der syntaktischen Position eines primary (*row1* steht vor ( )). In i) und j) steht *f* in der syntaktischen Position einer destination, sowie *row1* in a), *row2* in b) und *y* in g) und k); in k) dagegen steht *f* in der syntaktischen Position eines primary (*f* steht vor ( )), sowie *read* in d). In j) ist das unit *sin(a)/a* ein 'Prozedur-Rumpf', *sin(a)* und *a* sind darin aber Operanden und *sin* ist ein primary; in *sin(a)* ist *a* ein actual-parameter.

Mit seiner syntaktischen Position kennt man die Art des Kontextes, in dem ein bestimmtes unit steht und allein von der Art des Kontextes hängt es ab, welche Anpassungs-'Operatoren' durch den Compiler eingefügt werden können.
Will man also wissen, ob ein bestimmter Wert "aposteriori" von einem unit geliefert werden kann, so musz man die syntaktische Position dieses unit kennen und die 'Durchschlagskraft' dieser syntaktische Position, das heiszt: die Art des Kontextes.

4.4.2. Die verschiedene Arten des Kontextes

In ALGOL68 gibt es fünf verschiedene Arten von Kontext: (vgl.4.4.1)

1) "strong":

die rechte Seite einer assignation (eine source)
(*row1* in b, vgl. auch 1.3.1)

die rechte Seite einer identity-declaration (also auch eine source, vgl. 3.1.9) und damit dann auch die Position der aktuellen Parameter in einem call
(*row1* in c und d, *x* in k)

der 'Prozedur-Rumpf' innerhalb eines routine-text
(*sin(a)/a* in j, vgl. auch 3.1.5 und 3.2.2)

die Position nach dem declarer in einem cast
(*i* in h, vgl 4.1.1 und 4.4.3)

alle units die nicht das letzte unit sein (oder nicht einem completer vorangehen) in einer serial-clause; units also, die durch das "voiding" *void* liefern

2) "firm": die Position der Operanden in formulas
(d.h. links und rechts von einem dyadischen- und rechts von einem monadischen Operator, wie z.B. *row1* und *row2* in e und *row1*[*i*] und *vecrow*[*h*][*k*] in n, vgl. auch 5.1.3)

3) "meek": die enquiry-clause in conditional-, case- und conformity-clauses
(vgl. 2.6.1, 2.7.1 und 4.3.4)

das unit nach *from*, *by*, *to* und *while* in einer loop-clause (vgl. 2.8.1)

der Index bzw. die Grenzen innerhalb eines slice
(*h* und *k* in f, m und n, *i* in g, m und n)

der primary (die Position vor ( )) in einem call
(*sin* in j und *f* in k)

4) "weak": die Position rechts von dem *of* in einer selection
(*henry8* in l)

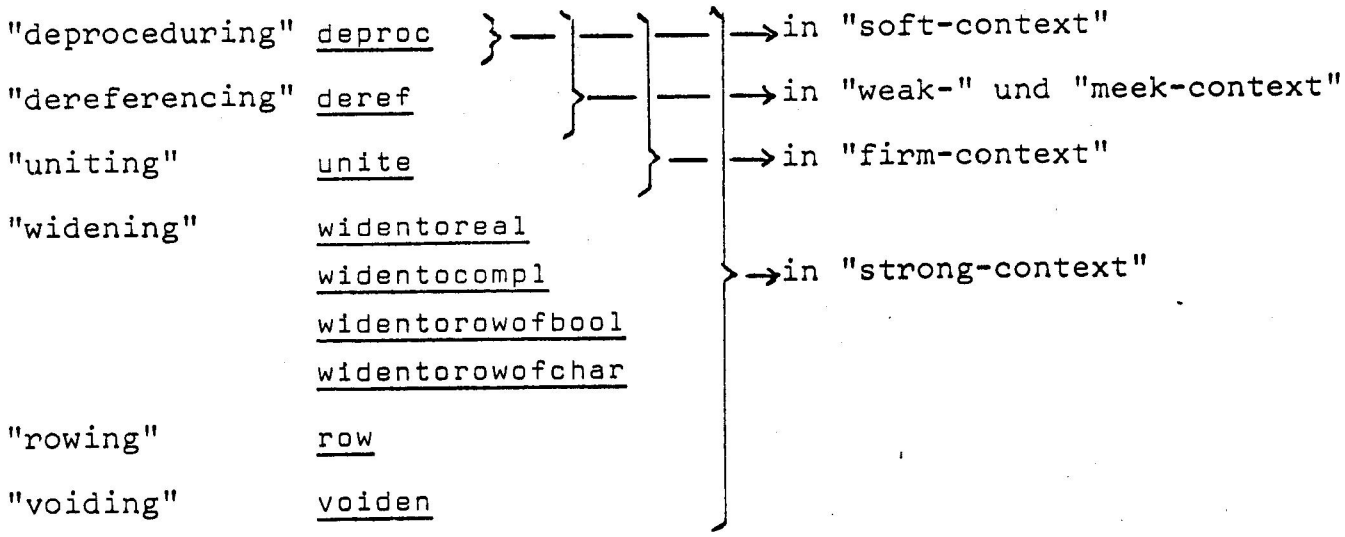
der primary (die Position vor [ ]) in einem slice
(*row1* in f, g und n, *vecrow* in m und n)

5) : die linke Seite (destination) einer assignation.

Gerade hier wird die ALGOL68-Syntax recht kompliziert und dazu kommt dann noch, dasz die Art des Kontextes durch eine "coercion" auch noch geändert werden kann.

Der Programmierer braucht sich dieser Komplikationen kaum (und meistenfalls gar nicht) bewuszt zu sein: die ALGOL68-Syntax ist gerade deswegen so kompliziert, weil sie mit gröszter Mühe so beschaffen ist, dasz sie genau alles dasjenige erledigt, was der Programmierer intuitiv erwartet.

Im nachfolgenden Überblick sehen wir, welche "coercions" in welcher Art von Kontext durch den Compiler eingefügt werden können, damit der Programmierer sich darüber den Kopf nicht zu zerbrechen braucht:



Das Wichtigste, was man diesem Überblick entnehmen kann, ist die Tatsache, dasz das "widening" (ebenso wie das "rowing") im "firm-context" nicht getan wird (also nicht in formulas) und dasz alle sechs "coercions" im "strong-context" getan werden können. Man soll auch nicht vergessen, dasz in einer destination (also "soft-context") nur "deproceduring" möglich ist.

Das Allerwichtigste ist jedoch, dasz man (im Zweifel) immer ein "strong-context" forcieren kann durch Anwendung des cast.

ALGOL 68

EINFÜHRUNG IN DIE ALGORITHMISCHE
SPRACHE ALGOL68

deel C

REPRODUKTIE
FYSISCH LABORATORIUM

3. MODE UND IDENTIFIER

3.1 Die 'mode-maker' *ref* , ( ) , *struct* und *proc*

3.1.1 *nonref* , *refmod* und *amode*
3.1.2 Reihen ("multiple values")
3.1.3 Slices, indexers, trimmers, , und
3.1.4 Strukturen ("structured values")
3.1.5 Routinen
3.1.6 Definierbarkeit und Gleichheit von "modes"
3.1.7 Slices und selections
3.1.8 Der äuszere Aufbau einer identity-declaration
3.1.9 Die "elaboration" einer identity-declaration

3.2 *nonref*-declarations

3.2.1 Deklaration von *nonref*-identifiers
3.2.2 Deklaration von Prozeduren
3.2.3 Prozeduraufruf (ohne Parameter)
3.2.4 Prozeduraufruf ('call-by-value')

3.3 *refmod*-declarations

3.3.1 Deklaration von *refmod*-identifiers
3.3.2 Prozeduraufruf ('call by reference')
3.3.3 Routinen die einen *refmod*-Wert liefern

3.4 "Generators"

3.4.1 *loc* und *heap*
3.4.2 Deklaration von Variablen
3.4.3 Initialisierung von Variablen
3.4.4 Deklaration von *ref* *procmod*-identifiers
3.4.5 Rekursive Routinen

3.5 Weitere *refmod*-declarations

3.5.1 Das Konzept eines "pointers"
3.5.2 Der cast
3.5.3 Die identity-relation
3.5.4 Reihen und Strukturen mit *refmod*-Elementen

3.4. Generators

3.4.1. *loc* und *heap*

Die bisher behandelten identity-declarations erfüllten stets zwei Aufgaben:

1) durch die "elaboration" einer identity-declaration erhält der in ihr auftretende identifier einen bestimmten Wert;

2) es wird (schon für den Compiler) festgelegt, welchen "mode" dieser Wert des identifier hat.

Von einer identifier-declaration erwartet man darüberhinaus oft noch ein drittes:

3) die Reservierung von Speicherplatz zur Aufnahme von Werten.

Dies ist einem von anderen Programmiersprachen her geläufig, so zum Beispiel in ALGOL60, wenn man schreibt *real x,y;* .

Die Reservierung von Speicherplatz geschieht in ALGOL68 durch Erzeugung einer neuen Variablen im Speicher. Für diese Erzeugung gibt es spezielle units, nämlich die generators. Hiervon gibt es zwei Arten:

a) local-generator : *loc amode*

b) heap-generator : *heap amode*

Dementsprechend gibt es in ALGOL68 zwei Möglichkeiten, den Speicher zu verwalten:

a) "stack"

b) "heap"

Dabei wird der "stack" nach der Methode des "Last-In-First-Out" ("LIFO") verwaltet: die zuletzt auf dem "stack" erzeugte Werte sind gerade diejenigen, die man zuerst wieder vergessen kann. Der "heap" dagegen wird so verwaltet, dasz in(!) ihm möglicherweise Werte länger erreichbar bleiben, als andere Werte, die früher erzeugt wurden (in diesem Speicherbereich können also Löcher entstehen: Zwischenräume von Werten, die nicht mehr benötigt werden).

Der local-generator erzeugt also eine Variable auf dem "stack", der heap-generator dagegen verwendet die raffiniertere (aber in den meisten Implementationen leider auch zeitaufwendigere) Verwaltung des "heap".

Die Lebensdauer (also der "scope", vgl. 1.2.6) der Adresse der durch den local-generator auf dem "stack" erzeugten Variablen ist identisch mit der Bearbeitungszeit der kleinsten serial-clause, in der diese Erzeugung stattfand.
Im Gegensatz dazu existiert die Adresse der auf dem "heap" mittels eines heap-generator erzeugten Variablen immer so lange, wie sie innerhalb des Programms benötigt wird. Dies kann durchaus noch nach der Abarbeitung der kleinsten serial-clause, in der diese ("heap"-)Variable erzeugt wurde, der Fall sein. Wie man erreichen kann, dasz eine ("heap"-)Variable für längere Zeit im Speicher existiert, wird zum Beispiel in 3.5.4 gezeigt. Da in dem "heap" zur Speicherbereinigung (Löcherbeseitigung) recht aufwendige Techniken der 'garbage-collection' ('Müll-Aufsammlung') angewendet werden müssen, so wird man heap-generators nur dort verwenden, wo sie tatsächlich unvermeidlich sind.

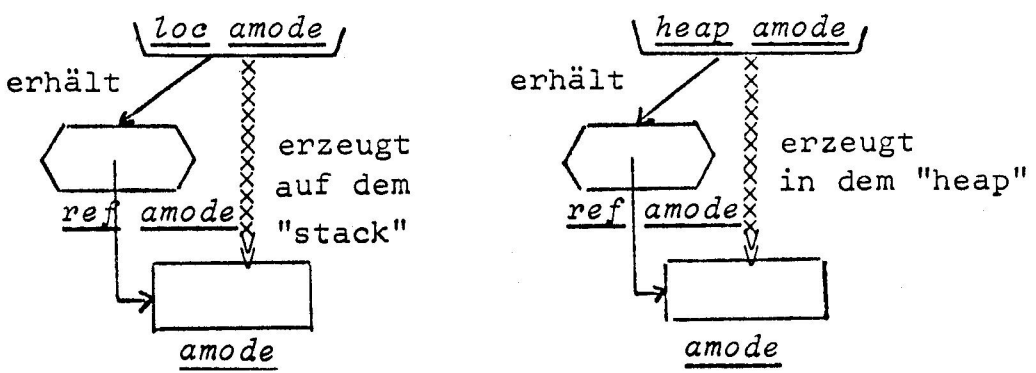
Wichtig für uns ist, dasz die generators *loc amode* und *heap amode* sich nur darin von einander unterscheiden, dasz durch sie Variable in verschiedenen Speicherbereichen erzeugt werden; in jeder anderen Hinsicht besteht zwischen ihnen kein Unterschied.

Die "elaboration" eines generator (*loc amode* bzw. *heap amode*) leistet stets zweierlei:

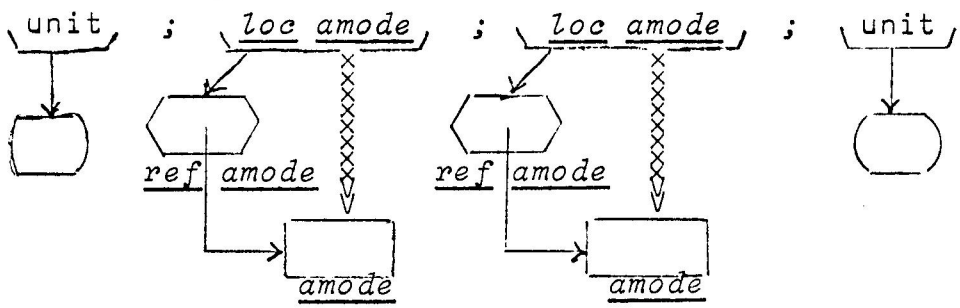
3a) es wird ein "locale" für einen *amode*-Wert erzeugt (auf dem "stack' bzw. in dem "heap") ,

3b) der generator selbst (als externes Objekt, also als unit) erhält die Adresse dieses erzeugten "locale" .

Dies können wir im Bild folgendermaszen darstellen:



Ein generator ist ein sehr dynamisches unit, das man von der Wirkungsweise her etwa mit einem Schaufelbagger vergleichen kann: wird die "elaboration" eines generator mehrmals hintereinander vorgenommen, so wird jedesmal aufs neue Baugrund (ein "locale") geschaffen. Weil ein generator ein unit ist, so könnte man sogar schreiben:



Dabei kann man jedoch mit dem Schauffelbagger (also dem generator) allein nicht viel anfangen; der von ihm geschaffene Baugrund ("locale") liegt brach und kann nicht genutzt werden, weil es ja keinen Bauherren gibt, der sich für die Adresse des neuen "locale" interessiert. Nach dem Passieren des go-on-symbol gibt es kein externes Objekt, das zu der Adresse der gerade eben erzeugten Variablen Zugriff hat; sie wird nicht weiter betrachtet (sogar weggeworfen), man vergleiche hierzu 4.1.5.
Wie man den geschaffenen Baugrund nutzen kann, wird in dem nächsten Paragraphen gezeigt.

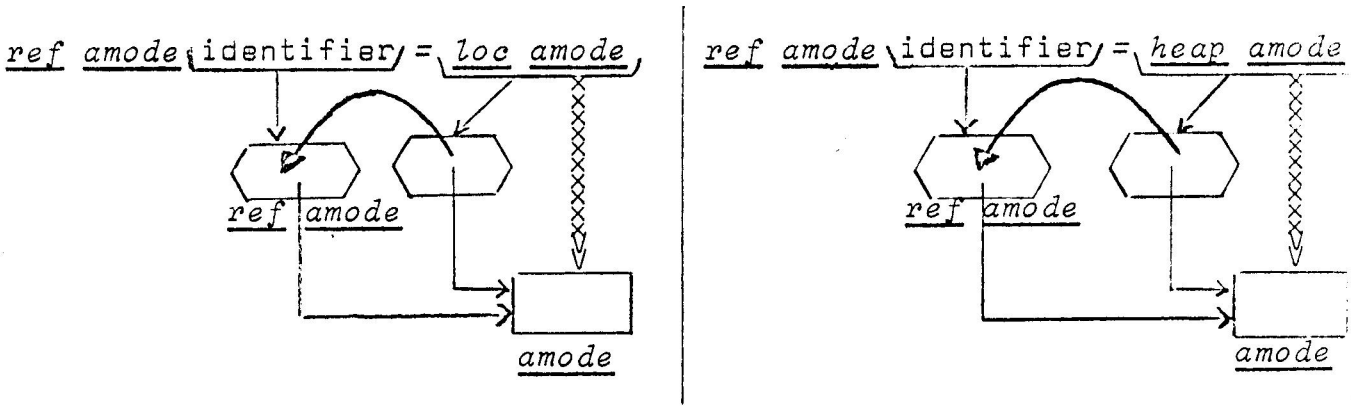
3.4.2. Deklaration von Variablen

Wie wir bereits in 3.1.8 gesehen haben, nennen wir eine identifier-declaration eine generating-identity-declaration, wenn das unit auf der rechten Seite ein generator ist. Da nun ein generator stets einen *ref amode* liefert, so hat solch eine generating-identity-declaration stets die Form:

ref amode identifier = *loc amode*

bzw. *ref amode* identifier = *heap amode*

Nun kann also der geschaffene Baugrund genutzt werden (der identifier tritt hier als 'Bauherr' auf), wie die "elaboration" zeigt:



Es hat demnach der identifier Zugriff ("access") zu der Adresse der durch den generator erzeugten Variablen (auf dem "stack" bzw. in dem "heap").
Wie wir in 3.1.1 verabredet haben, ist also der so deklarierte identifier (als externes Objekt) eine *amode*-Variable (also ein *ref amode*-identifier). Dieser Variablen können dann in nachfolgenden assignations *amode*-Werte zugewiesen werden.

Will man für mehrere Variable jeweils ein "locale" zur Aufnahme von Werten desselben "mode" erzeugen, so kann man dies in einer einzigen generating-identity-declaration tun, zum Beispiel:

ref real x = loc real , y = loc real

Hierdurch hat man also zwei *real*-Variablen *x* und *y* deklariert.

Diese Schreibweise ist jedoch für den Programmierer recht unbequem (besonders bei Deklaration mehrerer Variablen), daher hat man in ALGOL68 die Möglichkeit (wie wir schon oft gesehen haben), eine Variable auch in einer variable-declaration zu deklarieren:

loc *amode* identifier oder kürzer: *amode* identifier

bzw. *heap* *amode* identifier

Mehrere Variable können dann auch in einer einzigen variable-declaration deklariert werden, zum Beispiel:

real *x* , *y*

statt

ref *real* *x* = *loc* *real* , *y* = *loc* *real*

In dem Fall, wo man für mehrere Variable in einer einzigen identifier-declaration jeweils ein "locale" erzeugen will, besteht ein bemerkenswerter Unterschied in der "elaboration" einer generating-identity-declaration und der entsprechenden variable-declaration:

> wenn man mehrere "locales" zur Aufnahme von "multiple values" erzeugen will, so ist man nur bei Verwendung der variable-declaration sicher, dasz gleichviele Elemente in den jeweils erzeugten "locales" aufgenommen werden können.

Man vergleiche (es sei deklariert *int* *n* := *10*) :

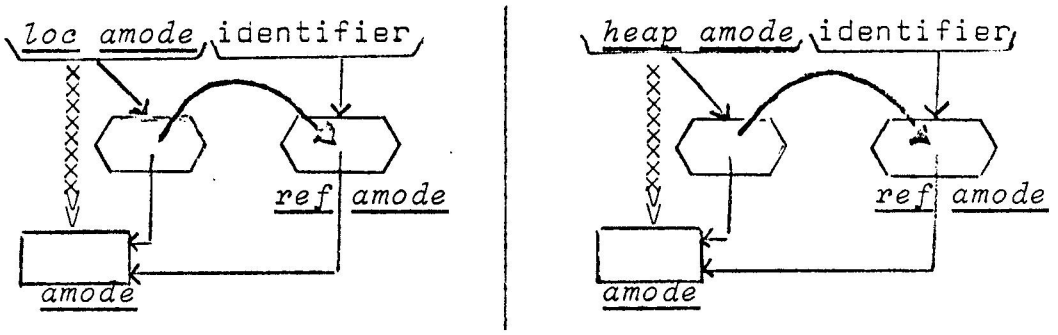
(a) (*1* : *n* *plusab* *1*) *real* *row1* , *row2* ;

und

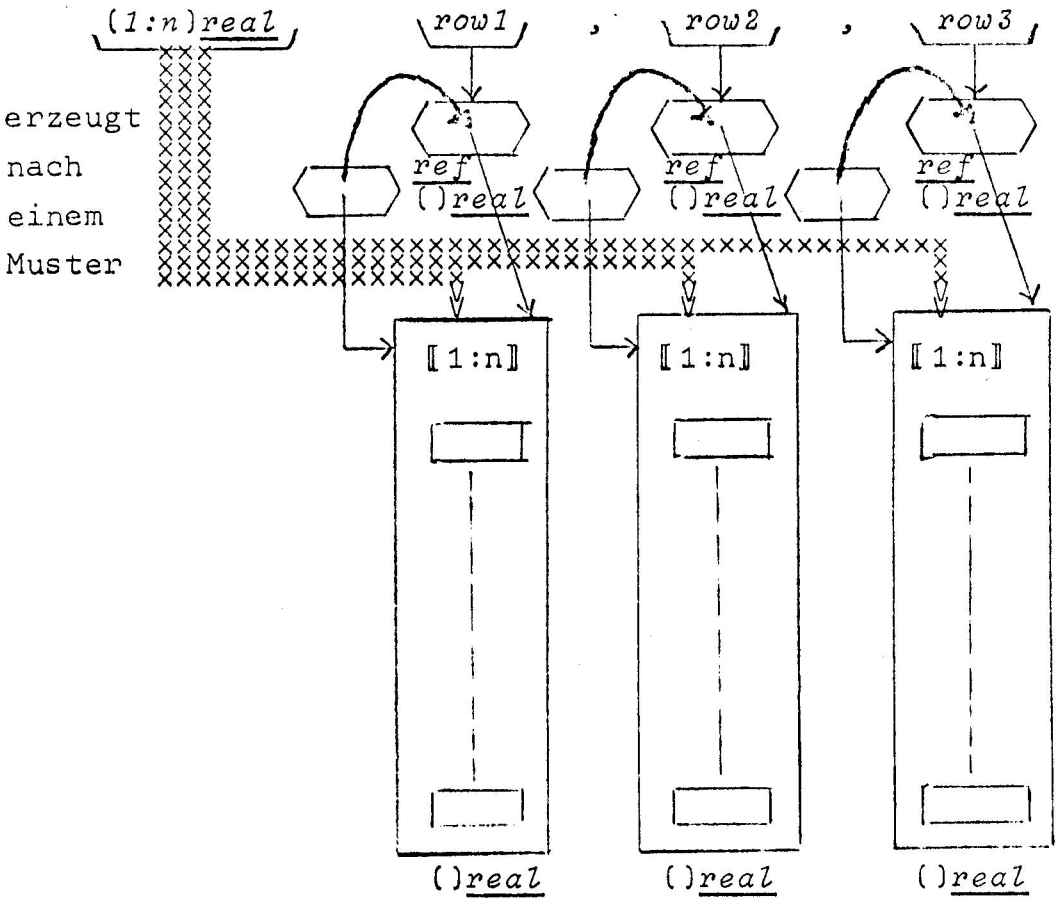
(b) *ref* ()*real* *row1* = *loc* (*1* : *n* *plusab* *1*) *real* ,
row2 = *loc* (*1* : *n* *plusab* *1*) *real* ;

Im Fall der variable-declaration (a) wird sowohl für *row1* als auch für *row2* ein "locale" erzeugt, welcher genau 11 Elemente (vom "mode" *real*) aufnehmen kann, der "descriptor" der erzeugten "multiple values" ist also jeweils ⟦1:11⟧.
Dagegen kann es im Fall (b) durchaus möglich sein (wegen der kollateralen Abarbeitung, vgl. 1.3.5), dasz die "descriptors" der erzeugten "multiple values" z.B. ⟦1:11⟧ bzw. ⟦1:12⟧, oder sogar ⟦1:12⟧ bzw. ⟦1:11⟧ oder beide ⟦1:12⟧ sind.

Die "elaboration" einer variable-declaration können wir auf folgende Weise in einem Bild darstellen:



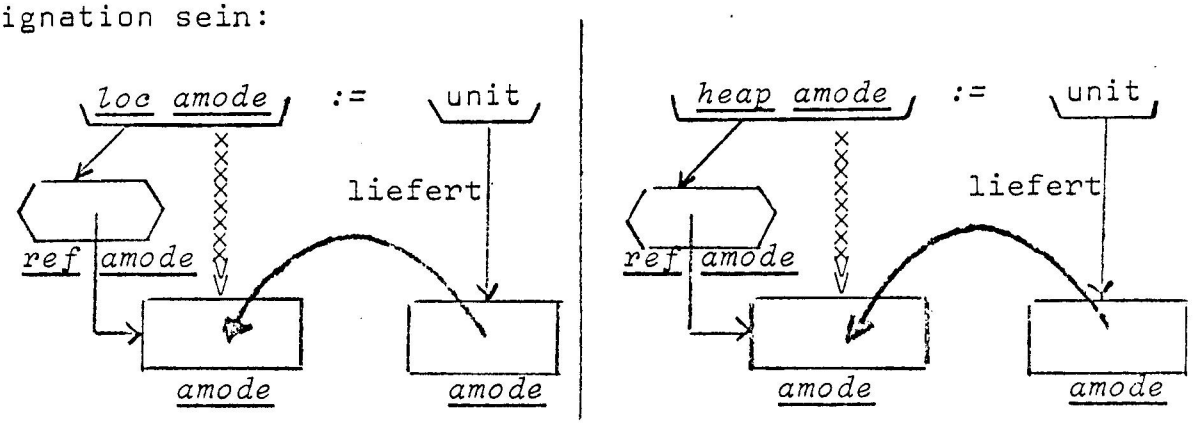
Betrachten wir dann folgendes Beispiel, indem wir für *amode* den declarer *(1:n)real* einsetzen und statt einen zugleich drei identifier deklarieren:



Hier werden also durch einen einzigen generator (*loc* *(1:n)real*) die erforderlichen "locales" im Speicher erzeugt. Alle "locales" richten sich nach einem Muster (der "descriptor" wird hier also nur einmal berechnet), daher wird so ein generator im Report als sample-generator bezeichnet.

3.4.3. Initialisierung von Variablen

Da ein generator stets die Adresse einer im Speicher erzeugten Variablen erhält, so kann er selbst die destination einer assignation sein:



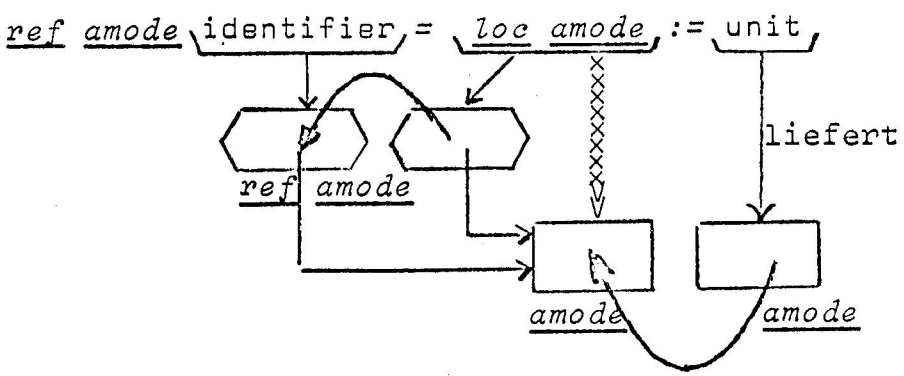
Solch eine assignation ist zwar syntaktisch (und auch semantisch) korrekt, aber man kann mit ihm allein wenig anfangen: in den neu erzeugten "locale" wird (aus einem anderen "locale") ein bestimmter Wert transportiert , jedoch gibt es keinen identifier, der die Adresse des neu erzeugten "locale" erhält. Die ganze assignation hängt in der Luft, so wie das alleinige Auftreten eines generator im Programmtext (vgl. 3.4.1).
Wir erinnern uns aber, dasz eine assignation selbst einen Wert liefert (vgl. 1.3.1): die Adresse, die ihre destination liefert. Somit kann also eine assignation selbst als unit auf der rechten Seite einer identity-declaration auftreten:

ref *amode* identifier = *loc* *amode* := unit

bzw.

ref *amode* identifier = *heap* *amode* := unit

Nun haben wir eine vernünftige Konstruktion erhalten, wie ihre "elaboration" zeigt:



Wir haben somit den identifier als eine Variable deklariert und ihr darüberhinaus gleich einen bestimmten Wert zugewiesen. Auch hier kann man mehrere Variable in einer einzigen identity-declaration mit Werten desselben "mode" initialisieren, also zum Beispiel durch

ref real x = loc real := 3.1415 , y = loc real := 2.7183

Für diejenigen,die sich noch immer darüber wundern, dasz eine assignation als Wert die Adresse ihrer destination liefert, sei hier angemerkt: die Initialisierung von neudeklarierten Variablen (so wie eben angegeben) ist nur möglich, da die assignation auf der rechten Seite der identity-declaration eine Adresse liefert.

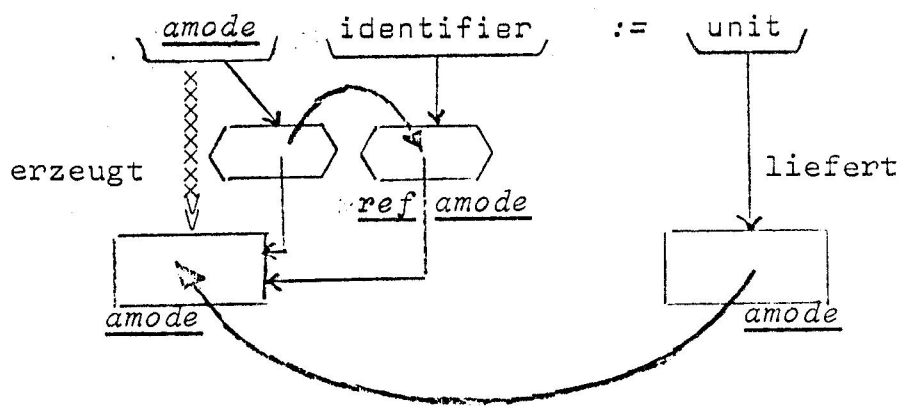
Da die Initialisierung von neudeklarierten Variablen nach der bisher betrachteten Weise für den Programmierer jedoch viel Schreibarbeit bedeutet, so kann man die Initialisierung auch in einer variable-declaration vornehmen:

loc amode identifier := unit oder *amode* identifier := unit

bzw.

heap amode identifier := unit

Ihre "elaboration zeigt folgendes Bild:



Beispiele:

int i := 0 ; real x := 3.14152653589 ;
(1:4)real little row := (0,0,0,0) ; compl z := (x,y) ;

Auch hier können in einer einzigen variable-declaration mehrere Variable initialisiert werden, zum Beispiel:

real x := 3.1415 , y := 2.7183 , u := 0.0 , v := 1.0 ,
eps := 0.1e-300 ;

Dasz das Auftreten eines generator und auch einer assignation der Form *loc amode* := unit, bzw. *heap amode* := unit nicht nur innerhalb einer identity-declaration sinnvoll ist, zeigt folgendes Beispiel:
Wir haben in 3.3.3 eine Prozedur deklariert, die die Adresse des grösten Elementes eines ()*real* liefert und überdies einem aktuell mit übertragenen *ref int* den Index ("subscript") dieses grösten Elementes zuweist. Diese Prozedur *max* kann man auch dann aufrufen, wenn man auszerhalb der Routine an dem Index des grösten Elementes gar nicht interessiert ist. Man schreibt dann einfach z.B:

```
max ( myrow , loc int ) := 0
```

Die Parameterübergabe wird dann vorgenommen durch die identity-declaration:

```
ref int imax = loc int
```

Wir können nun die Prozedur noch so abändern, dasz von der Stelle an gesucht wird, die durch die aktuell übertragene Variable bestimmt wird:

```
proc newmax = (()real row , ref int indx) ref real:
                ( for i from indx+1 to upb row
                  do
                     if row(i)>row(indx) then indx:= i fi
                  od ;
                  row(indx)
                ) ;
```

Will man nun von *k* ab suchen, will man jedoch *k* nicht ändern und ist man auch nicht interessiert an dem Index des grösten Elementes, so kann man zum Beispiel schreiben:

```
newmax ( myrow , loc int := k )  :=  0
```

Hier wird dann die Parameterübergabe vorgenommen durch die identity-declaration:

```
ref int indx = loc int := k
```

Diese Möglichkeit der Verwendung eines generator als aktueller Parameter ergibt sich ganz zwangsläufig aus dem "orthogonal design" der SPrache.

3.4.4. Deklaration von *ref procmode* identifier

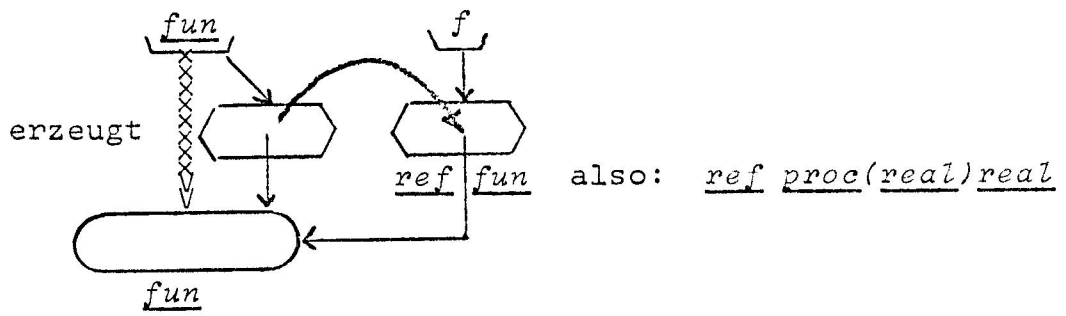
Da wir in ALGOL68 generell *amode*-Variable deklarieren können, so können wir also z.B. für *amode* auch einen *ref procmode*-declarer einsetzen.
Betrachten wir dazu die folgende mode-declaration:

mode fun = proc(real)real

Dann können wir eine *fun*-Variable deklarieren:

fun f

Ihre "elaboration" zeigt folgendes Bild:



Hier wird also durch den (sample-)generator ein "locale" zur Aufnahme von Routinen (vom "mode" *fun*) erzeugt. Der Speicherplatz für solch ein "locale" wird natürlich nicht so grosz, dasz er vollständige Routinen aufnehmen kann (diese sind ja zum Zeitpunkt der "elaboration" der variable-declaration noch gar nicht bekannt). Es braucht nur soviel Speicherplatz reserviert zu werden, wie für den Eintritt in die Routinen benötigt wird. So zum Beispiel Speicherplatz für die Parameterbehandlung sowie für die Adressen der Routinen, die selbst irgendwo im Objektcode zu finden sind. Bis auf einen festen, implementationsabhängigen Teil wird dieser Speicherplatz vollständig durch den "mode" der Routine (hier also *proc(real)real*) bestimmt.

Dem *f* kann man nun *fun*s zuweisen, z.B. die Routinen, die die Standard-Prozeduren *sqrt* und *sin* erhalten. Schreibt man dann die assignation

f := sqrt oder *f := sin*

so hat der nachfolgende call

f(x)

den gleichen Effekt wie

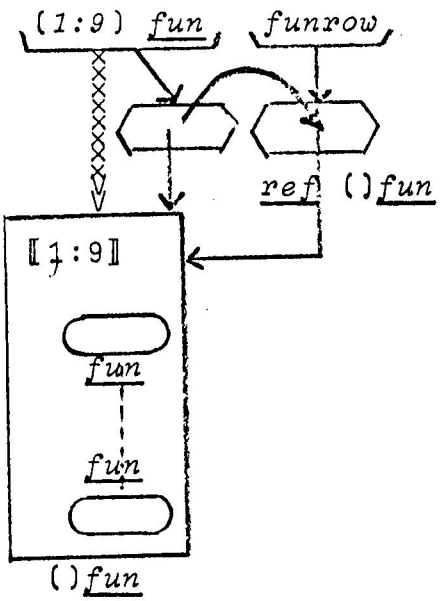
sqrt(x) oder *sin(x)*

Schreibt man etwa

f := *case* *i* *in* *sqrt* , *exp* , *ln* , *cos* , *arccos* ,
sin , *arcsin* , *tan* , *arctan*
esac

so ist der Effekt von *f(x)* klar.

Desweiteren kann man eine Reihe von *fun*s deklarieren, zum Beispiel:



Wir können dann zuweisen:

funrow := (*sqrt*,*exp*,*ln*,*cos*,*arccos*,*sin*,*arcsin*,*tan*,*arctan*)

Die source ist hier ein row-display von *fun*s .

Durch den call

funrow(*i*)(*x*)

wird sodann das i-te *fun*-Element aufgerufen mit dem aktuellen Parameter *x*.

3.4.5. Rekursive Routinen

Wenn eine Routine aufgerufen wird, so können bei der "elaboration" ihrer entsprechenden closed-clause (vgl. 3.2.4 und 3.3.2) weitere Routinen aufgerufen werden. In ALGOL68 (wie in fast allen modernen Programmiersprachen) besteht dabei die Möglichkeit, dasz auf direkte oder indirekte Weise eine Routine sich selbst wieder aufruft.

Wird dann solch eine rekursive (sich selbst aufrufende) Routine aufgerufen, so findet bei der "elaboration" ihrer closed-clause deren "elaboration" möglicherweise in verschiedenen Rekursionstiefen statt. Ihre "elaboration" wird also mehrmals in sich selbst vorgenommen. Man könnte auch sagen: die closed-clause kann sich selbst aufs neue beleben.

Betrachten wir nun, was bei solch einer Wiederbelebung in einer bestimmten Rekursionstiefe passiert:

Zunächst werden "locales" zur Aufnahme der an die Routine aktuell übertragenen sowie der für sie lokalen Werte auf dem "stack" erzeugt. Dabei verliert die closed-clause den Zugriff zu den in ihrem (noch nicht vollendeten) Vorleben erzeugten "locales". Hat nun die closed-clause ihr neues Leben einmal vollendet, so kehrt sie zu ihrem Vorleben zurück. Es verschwinden dann die zuletzt erzeugten "locales" von dem "stack", und sie findet die zu ihrem Vorleben gehörenden "locales" auf dem "stack" wieder vor.

Man beachte, dasz dieses Vorgehen beim Aufruf einer rekursiven Routine genau dem 'Last-In-First-Out'-Prinzip der "stack"-Verwaltung entspricht (vgl. 3.4.1). Der Programmierer jedoch braucht sich um diese Verwaltung nicht zu kümmern, sie wird ihm vom "runtime"-System abgenommen. Dies ist für ihn sehr angenehm, gerade bei der Lösung von Aufgaben, bei der er sonst die Organisation eines "stack" in seinem Programm selbst vornehmen müszte. Allerdings musz der Programmierer sich darüber im klaren sein, dasz der Aufruf einer rekursiven Routine zu erheblicher Ineffizienz führen kann (eben weil sie sich mehrmals selbst aufruft). Deshalb sollte er prüfen, ob er die gewünschte Prozedur nicht auf einfache Weise auch iterativ deklarieren kann, obwohl die rekursive Deklaration meistens die elegantere ist.

Zum Beispiel betrachten wir den Euklidischen Algorithmus zur Bestimmung des grösten gemeinsamen Teilers zweier ganzer Zahlen:

rekursiv:

```
proc euclid = (int m,n) int:
        if n=0
      then abs m
      else euclid ( n , m mod n )
        fi ;
```

iterativ:

```
proc euclid = (int m,n) int:
      ( int dividend:=m, divisor:=n ;
      while int remainder = dividend mod divisor ;
                  remainder ≠ 0
         do dividend:=divisor ;
              divisor:=dividend
         od ;
        divisor
      ) ;
```

Man sieht hier deutlich, wie die Erzeugung von "locales" sowie der Transport von Werten in diese "locales" bei der iterativen Formulierung explizit angegeben werden musz. Bei der rekursiven Formulierung dagegen geschieht dies völlig implizit.

Wer Lust hat, könnte einmal versuchen die folgende rekursive Prozedur iterativ zu deklarieren:

```
proc ackermann = (int m,n) int:
        if m=0 then n+1
      elif n=0 then ackermann ( m-1 , 1 )
               else ackermann ( m-1 , ackermann ( m , n-1 ) )
        fi ;
```

Es lohnt sich jedoch mehr, darüber nachzudenken, wie sich *ackermann* selbst belebt zum Beispiel durch den Aufruf

```
ackermann ( 6,8 )
```

3.5. Weitere *refmod*-Deklarationen

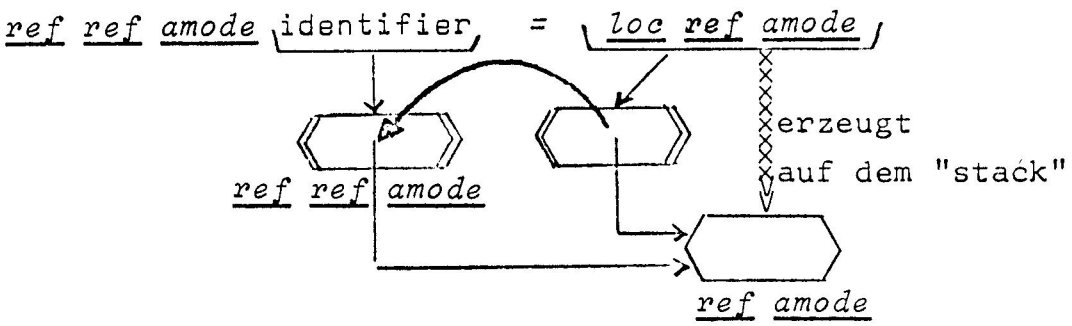
Hat man eine Variable deklariert, deren Adresse sich auf eine Struktur mit *ref amode*-fields bezieht, so gelangt man bei der Auswahl dieser fields auf natürlicher Weise auf eine höhere *ref*-Stufe (vgl. 3.1.4). In diesem Paragraphen wollen wir uns mit solchen höheren *ref*-Stufen auf mehr systematische Weise beschäftigen.

3.5.1. Das Konzept eines "pointers"

Betrachten wir die generating-identity-declaration

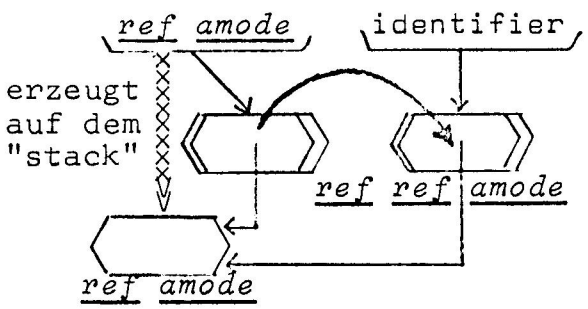
ref ref amode identifier = *loc ref amode*

Es wird hier also auf dem "stack" ein "locale" zur Aufnahme einer (*ref amode*) Adresse erzeugt. Der identifier erhält dann die Adresse dieses "locale":

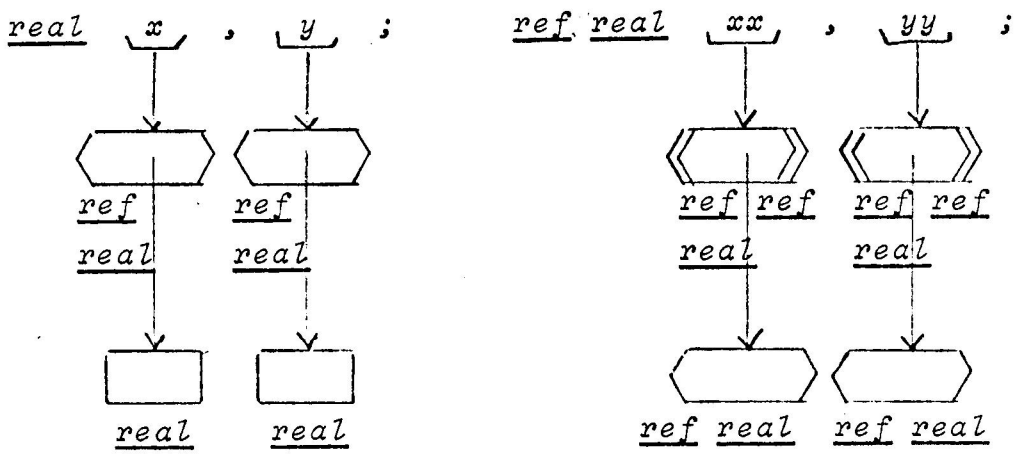


Wir erhalten also eine Variable auf einer höheren *ref*-Stufe. Solch eine Variable wird oft als "pointer" bezeichnet.

Den gleichen Effekt können wir selbstverständlich (und auch kürzer) erreichen durch die variable-declaration



Sei zum Beispiel folgendes deklariert:



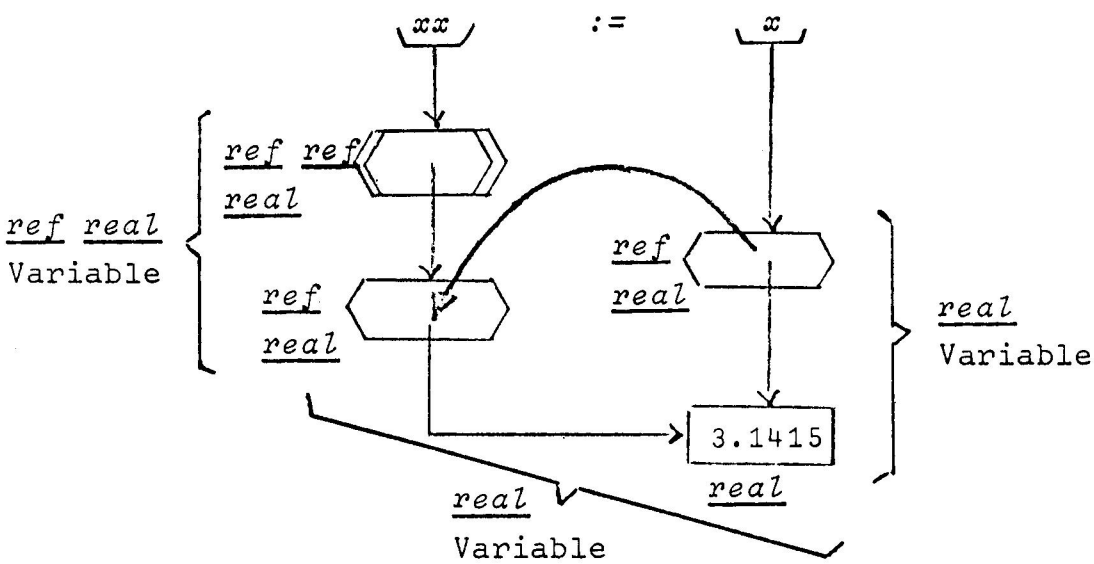
Betrachten wir nun nach

x := 3.1415

die folgende assignation

xx := x

so ergibt sich folgendes Bild:



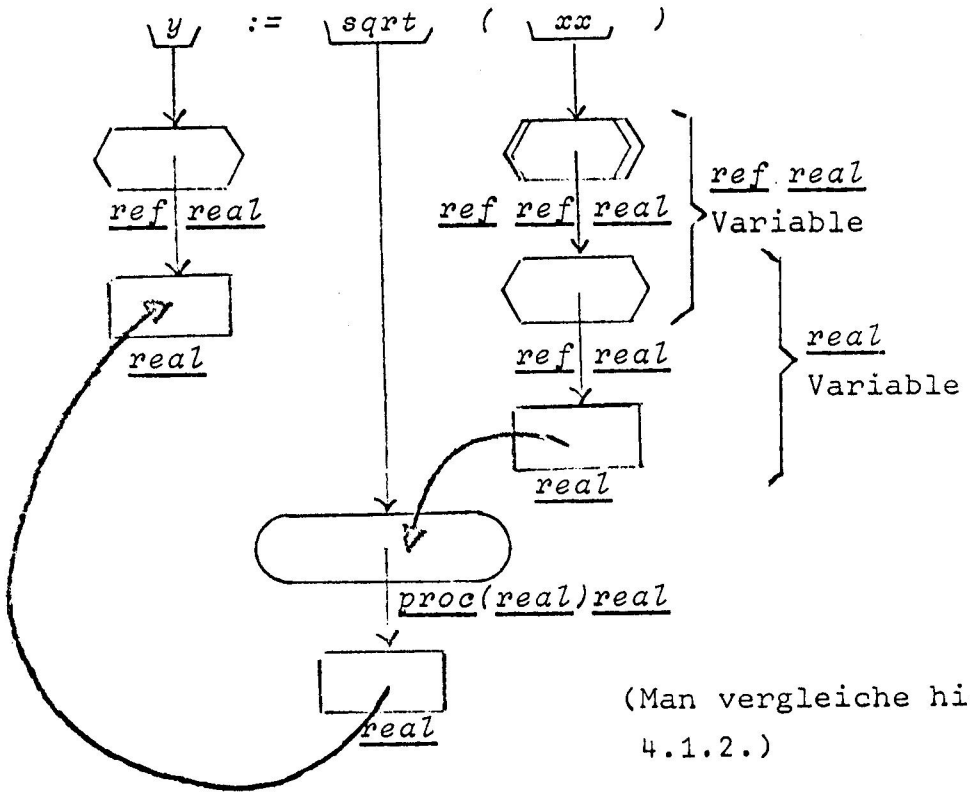
Also, *xx* erhält ein *ref ref real*, das sich auf ein *ref real* bezieht. Wir begegnen hier , Variablen auf zwei verschiedenen *ref*-Stufen: eine *ref real*-Variable (ein "pointer") und eine *real*-Variable.

Dies wird oft als indirekte Adressierung bezeichnet. Man beachte hierbei, dasz in eine assignation (so auch in der obigen *xx := x*) der Wert-Transport immer auf der höchstmöglichen *ref*-Stufe (diese wird stets durch die destination bestimmt) stattfindet.

Betrachten wir dann weiter

y := sqrt(xx)

Hier wird der Adresse von *y* der *real*-Wert zugewiesen, den die Routine von *sqrt* bei ihrem Aufruf liefert:



(Man vergleiche hier 4.1.2.)

Nach der assignation *xx := x* ist

y := sqrt(xx)

gleichbedeutend mit

y := sqrt(x)

Hätten wir jedoch zuerst geschrieben

xx := y

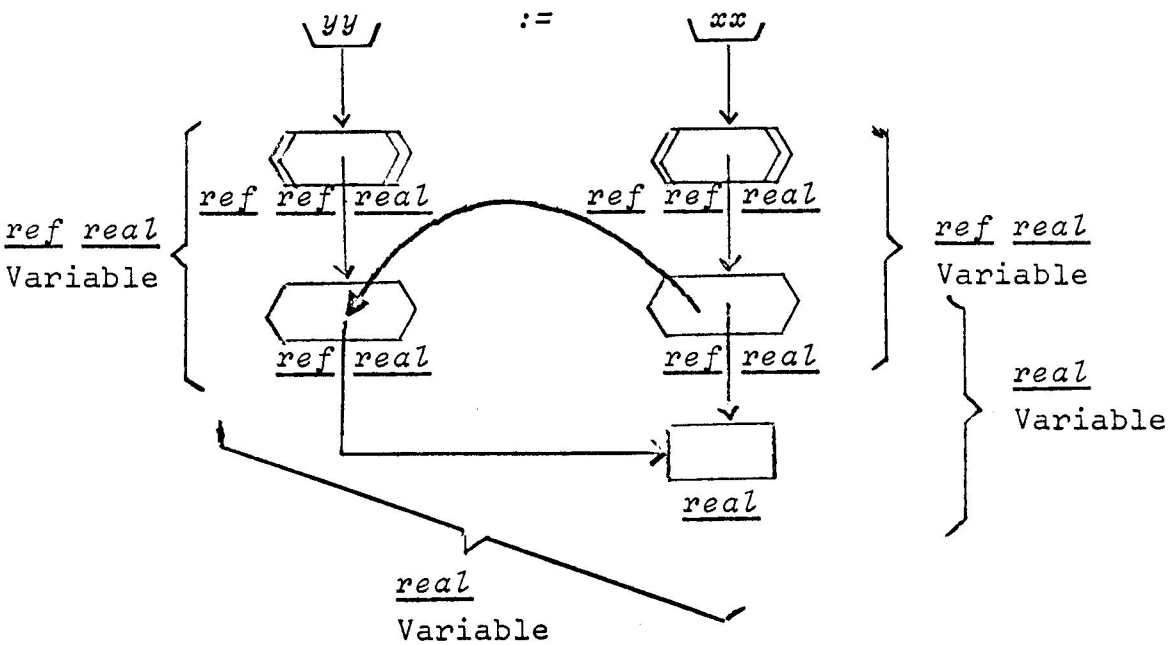
so wäre dagegen

y := sqrt(xx)

gleichbedeutend mit

y := sqrt(y)

Betrachten wir zum Schlusz noch die assignation



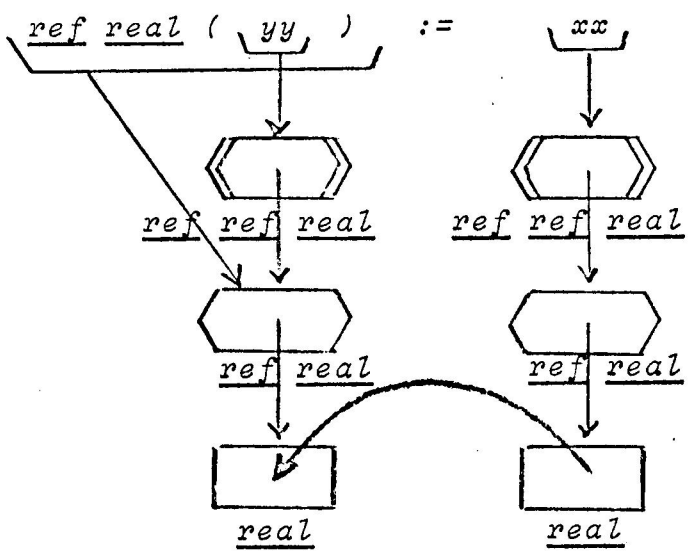
Man beachte wieder, wie auch hier der Wert-Transport auf der höchsten *ref*-Stufe vorgenommen wird.

3.5.2. Der cast

Wie wir in 3.5.1 gesehen haben, wird durch

yy := xx

dem *ref ref real* von *yy* ein *ref real* zugewiesen (eine *real*-Variable).
Will man jedoch, dasz der Wert-Transport auf einer *ref*-Stufe niedriger vorgenommen wird, so müssen wir die destination *yy* zwingen, einen *ref real* zu liefern. Wie wir schon in 2.2.2 gesehen haben, können wir dies mit Hilfe des cast (vgl. 4.4.3) erreichen:



Es wird hier keine Adresse transportiert, sondern es wird dem *ref real*, auf den sich der *ref ref real* (von *yy*) bezieht, ein *real*-Wert zugewiesen.
Dies könnten wir natürlich auch dadurch zum Ausdruck bringen, dasz wir schreiben:

ref real (yy) := real (x)

Die Benutzung des cast auf der rechten Seite einer assignation ist jedoch überflüssig, da von dem Kontext der assignation her schon klar ist, was geschehen soll (vgl. 4.1.2 und 4.4.3).

Zum guten Verständnis beachte man noch, dasz nach den assignations

yy := x ; xx := y

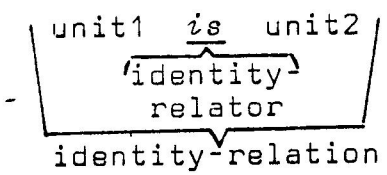
die obige assignation *ref real (yy) := xx* gleichbedeutend ist mit

x := y (!)

3.5.3. Die identity-relation

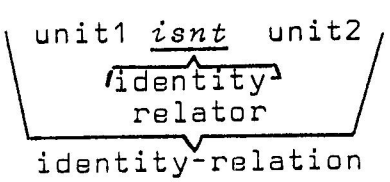
Da in ALGOL68 Adressen selbst Werte sein können, so kann man fragen, ob zwei Adressen, die sich auf Werte desselben "mode" beziehen, gleich sind oder nicht.
Zur Beantwortung dieser Frage können nicht die Operatoren "=" und "≠" benutzt werden, da durch sie (standardmäszig) immer nur *nonref*-Werte verglichen werden. Sie stehen deswegen nicht mehr für den Adressenvergleich zur Verfügung. Für den Adressenvergleich sind überhaupt keine Operatoren vorhanden (und auch nicht nötig), da er in der Sprache gegeben ist für Adressen von beliebigem "mode", genauso wie die Zuweisung (also der Wert-Transport) für Werte von beliebigem "mode" gegeben ist.

Für den Adressenvergleich haben wir dann die identity-relation. Ihre allgemeine Form ist entweder:



bzw. unit1 *:=:* unit2

oder

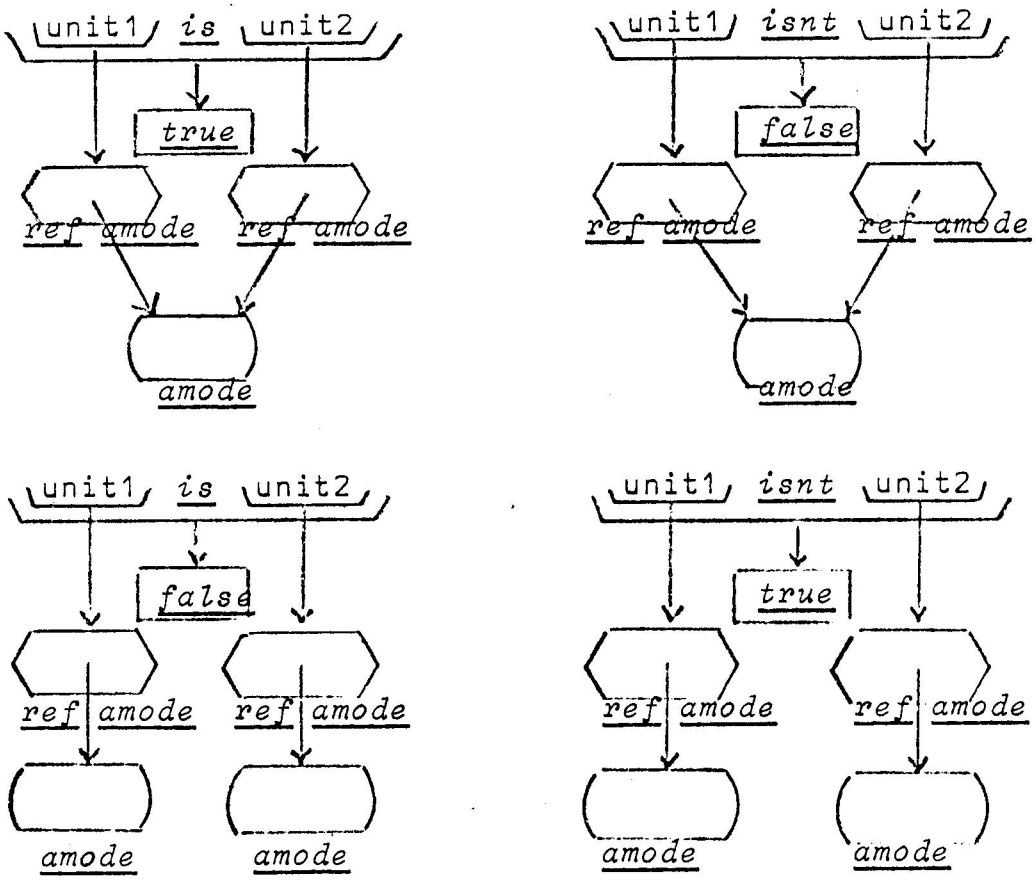


bzw. unit1 *:≠:* unit2

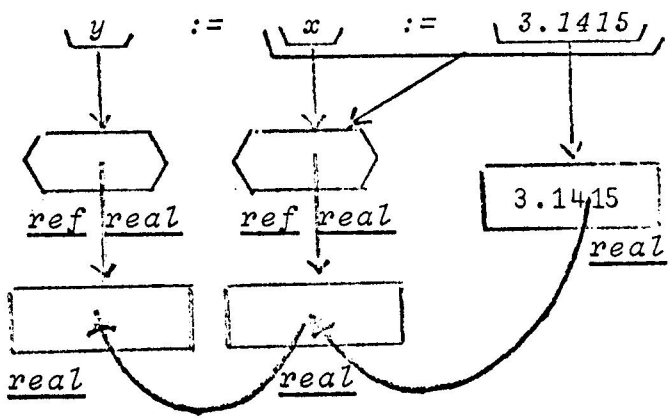
Hierbei müssen sowohl unit1 als auch unit2 jeweils einen *ref amode* liefern. Wenn sie dies nicht können, so ist die identity-relation syntaktisch inkorrekt und es erfolgt dann eine Fehlermeldung des Compilers.
Desweiteren wird der Adressenvergleich stets auf der höchstmöglichen *ref*-Stufe vorgenommen, auf der die Adressen denselben "mode" hab
Eine identity-relation liefert ein *bool*. Sind die beiden Adressen, die unit1 und unit2 bei ihrer kollateralen Abarbeitung liefern, gleich, so erhält die identity-relation der Form unit1 *is* unit2 den Wert *true*. Sind sie dagegen ungleich, so liefert sie in dieser Form *false*. Für die identity-relation mit *isnt* gilt genau das Umgekehrte.

Dabei sind zwei (*ref amode*) Adressen gleich, wenn sie beide die Adresse eines einzigen "locale" bestimmen, sich also auf denselben *amode*-Wert (innerhalb dieses "locale") beziehen.

Die "elaboration" einer identity-relation können wir im Bild folgendermaszen zum Audruck bringen:



Betrachten wir nun zum Beispiel die folgende assignation



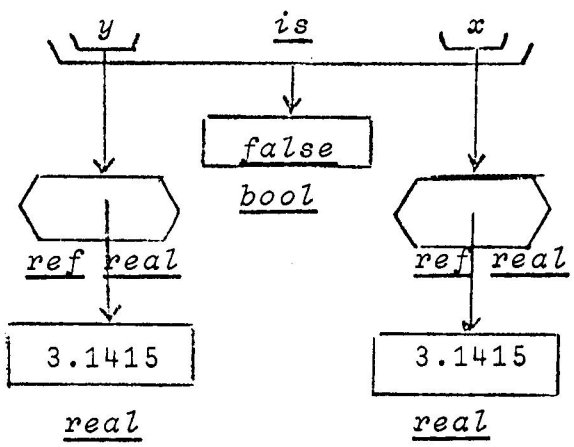
Nach dieser "elaboration" erhält dann zwar die formula

y = *x*

den Wert *true*, jedoch erhält die identity-relation

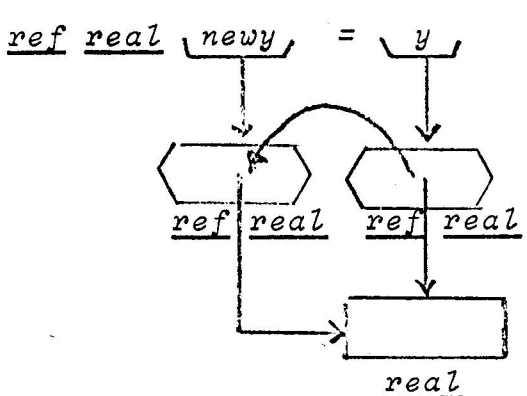
y *is* *x*

stets den Wert *false*, wie ihre "elaboration" zeigt:



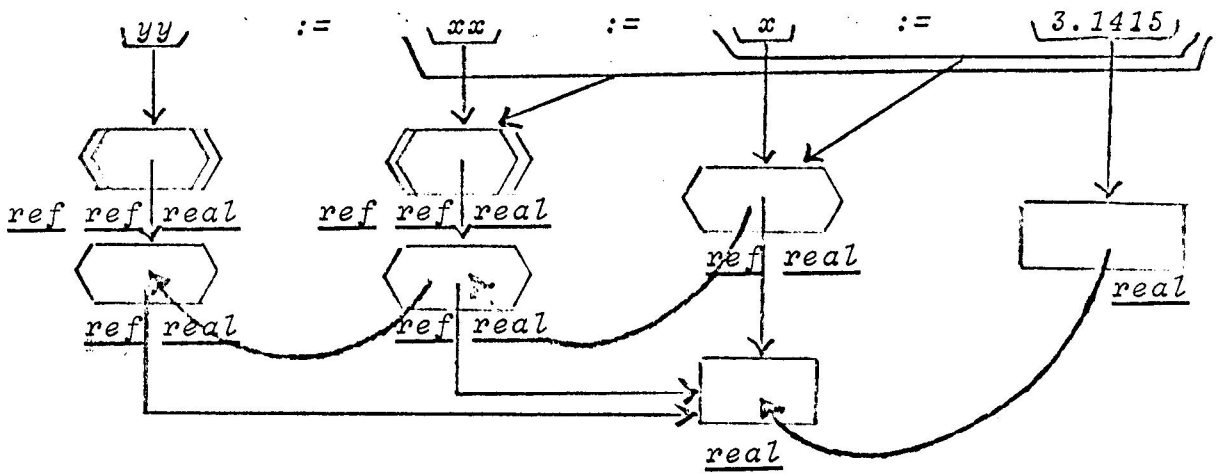
Die beiden (*ref* *real*) Adressen von *x* und *y* sind voneinander verschieden, obwohl die *real*-Werte, auf die sie sich beziehen gleich sind; denn diese sind in verschiedenen "locales" untergebracht. Die Werte, auf die sich die *ref* *real*s von *x* bzw. *y* beziehen, spielen absolut keine Rolle: die identity-relation *y* *is* *x* liefert *false*, weil durch die variable-declaration *real* *x*,*y* zwei verschiedene "locales" erzeugt wurden.

Bemerkenswert ist nun, dasz nach der identity-declaration



die identity-relation *newy* *is* *y* den Wert *true* liefert. Es wurde durch *ref* *real* *newy* = *y* kein neuer "locale" erzeugt.

Wir betrachten nun die assignation



Nun liefert zwar die identity-relation

yy *is* *xx*

den Wert *false*, jedoch liefern die identity-relations

xx *is* *x* und *yy* *is* *y*

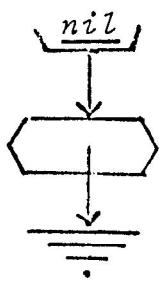
den Wert *true* (!), man vergleiche hierzu 4.4.4.

Es ist klar, dasz dann auch

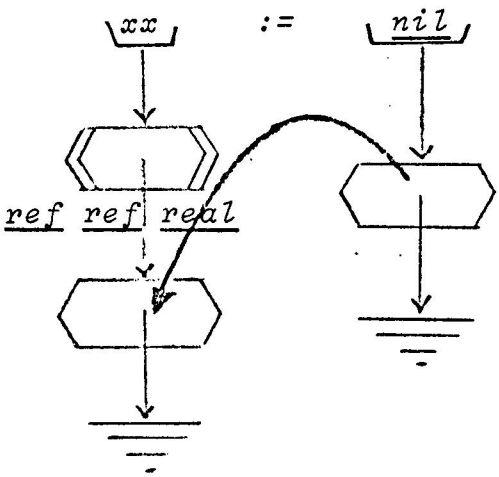
xx *is* *ref* *real* (*yy*) und *ref* *real* (*xx*) *is* *yy*

true liefern werden.

Da wir hier über den Vergleich von Adressen sprechen, ist es Zeit, das 'Nullelement' innerhalb der Menge aller möglichen Adressen kennenzulernen. Dies ist ein spezielles unit, nämlich *nil*. Es liefert stets eine Adresse, die sich jedoch auf keinen Wert bezieht. Dies stellen wir im Bild folgendermaszen dar:



Dem *nil* kann man keinen Wert zuweisen, wohl aber kann man jeder *refmod*-Variablen den Wert von *nil* zuweisen, also zum Beispiel:



Man kann nun mit Hilfe einer identity-relation feststellen, ob eine Adresse *nil* ist. Nach *xx := nil* liefert also

xx is nil

den Wert *true*.

Man beachte, dasz dann jedoch die assignation

ref real (xx) := 3.1415

obwohl syntaktisch korrekt, semantisch nicht möglich ist (man kann dem *nil* keinen Wert zuweisen, siehe oben). Sie wird also zu einer ("runtime"-)Fehlermeldung führen.

Zum Schlusz noch eine Bemerkung zur Syntax :

man beachte, dasz die identity-relators "*is*" und "*isnt*" (also auch ":=:" und ":≠:") stärker binden als das becomes-token ":=".

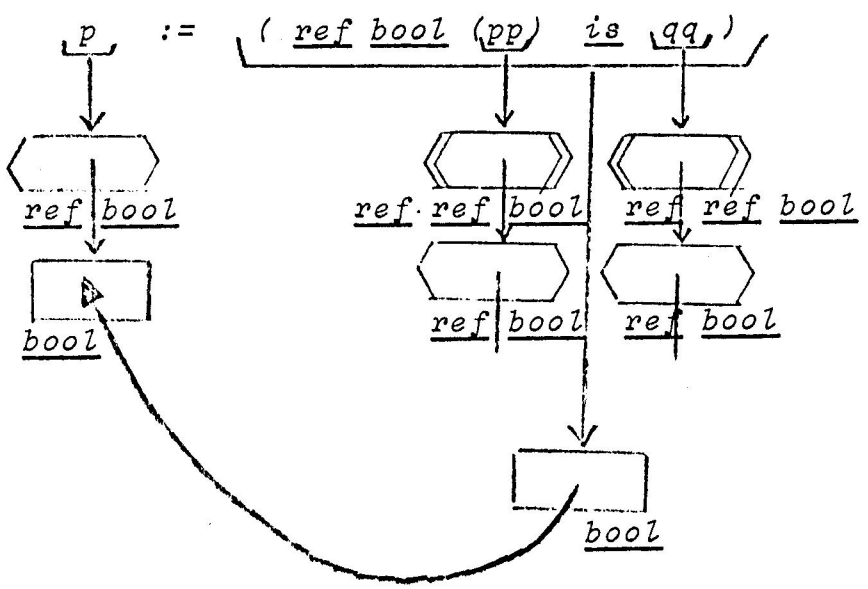
Haben wir zum Beispiel deklariert

bool p ; ref bool pp , qq ;

so kann man schreiben:

p := ref bool (pp) is qq

und es erhebt sich die Frage wie dieser Ausdruck (implizit) geklammert wird. Die Klammerung ist so, dasz dieser Ausdruck eine assignation ist und nicht eine identity-relation, also:



Dagegen ist auf jeden Fall der Ausdruck

p is qq := rr

syntaktisch inkorrekt . Die implizite Klammerung ist hier:

(p is qq) := rr

und die destination liefert hier immer einen *nonref*.

3.5.4. Reihen und Strukturen mit *refmod*-Elementen

Hat man mit Hilfe eines generator ein "locale" zur Aufnahme eines "multiple value" erzeugt, dessen Elemente selbst Adressen (also *ref amode*s) Sind, so gelangt man bei der Auswahl (durch ein slice, vgl. 3.1.2) dieser Elemente ebenfalls auf eine höhere *ref*-Stufe.

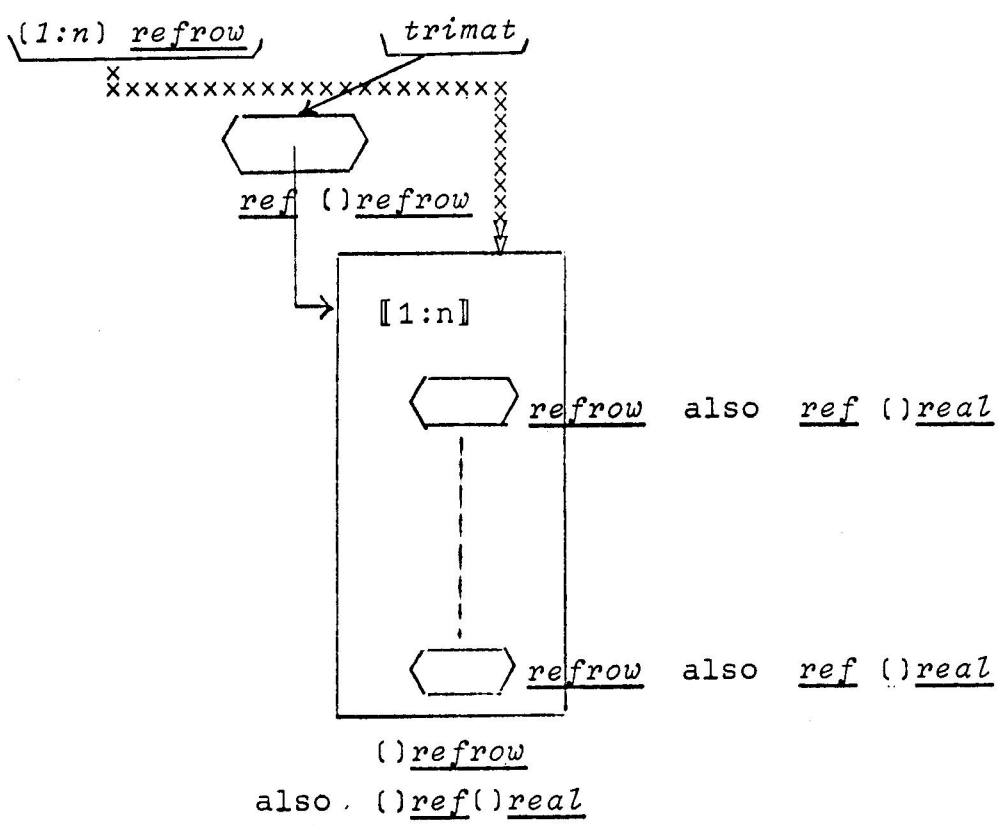
Betrachten wir dazu die folgende mode-declaration

mode refrow = ref ()*real*

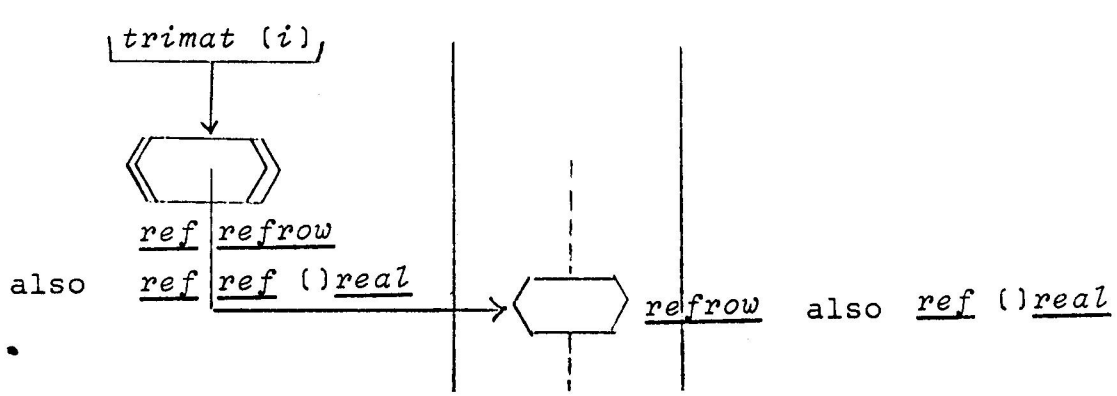
Deklarieren wir dann

(*1:n*)*refrow trimat*

so erhalten wir folgendes Bild:



Das i-te Element können wir dann auswählen durch den slice



Es liefert also *trimat*(*i*) ein *ref* *ref* ()*real*, dem wir die Adresse eines ()*real* zuweisen können. Dabei kann die Anzahl der Elemente solch eines ()*real* für jedes i ($1 \leq i \leq n$) durchaus verschieden sein.

Mit Hilfe zum Beispiel der loop-clause

for *i* *to* *n* *do* *trimat*(*i*) := *heap* (*1:i*)*real* *od*

werden n verschiedene "locales" zur Aufnahme von ()*reals* mit aufsteigender Anzahl von Elementen erzeugt, deren Adressen dann als Elemente des "multiple value" von *trimat* auftreten.

Durch die Erzeugung dieser n "locales" hat man insgesamt Speicherplatz zur Aufnahme der Elemente einer Dreiecksmatrix reserviert. Man beachte hierbei die Notwendigkeit des heap-generator: bei jeder Abarbeitung der serial-clause zwischen *do* und *od* wird ein weiterer "locale" in dem "heap" erzeugt. Da dann seine Adresse von dem "locale" des i-ten Elementes von *trimat* aufgenommen wird, so kann er auch auszerhalb der serial-clause weiter existieren; denn der "locale" zur Aufnahme solch einer Adresse ist auszerhalb der loop-clause erzeugt worden.

Durch leichte Abänderung der obigen loop-clause kann man auch noch erreichen, dasz die Elemente der Dreieckmatrix gleich initialisiert werden:

(*1:n*)*real* *o*; *for* *i* *to* *n* *do* *o*(*i*):=*0* *od*;

for *i* *to* *n* *do* *trimat*(*i*) := *heap* (*1:i*)*real* := *o*(*1:i*) *od* ;

Will man das (i,j)-te Element ($1 \leq i \leq n, 1 \leq j \leq i$) auswählen, so erreicht man dies natürlich durch Verwendung des slice *trimat*(*i*)(*j*) (man vergleiche 3.1.7).

Betrachten wir nun ein Beispiel für das Auftreten von *refmods* innerhalb von Strukturen.

Wir haben schon bei der Heirat von *henry8* und *anna boleyn* gesehen (vgl. 3.3.2), dasz man über "pointers" Strukturen miteinander verknüpfen kann. Sind dann die Strukturen alle vom gleichen "mode", so kann man sie auf einfache Weise zu einer Liste verketten.

Es sei deklariert *int n* = unit . Wir definieren nun

mode element = *struct* ((1:n)*real row* , *ref element next*)

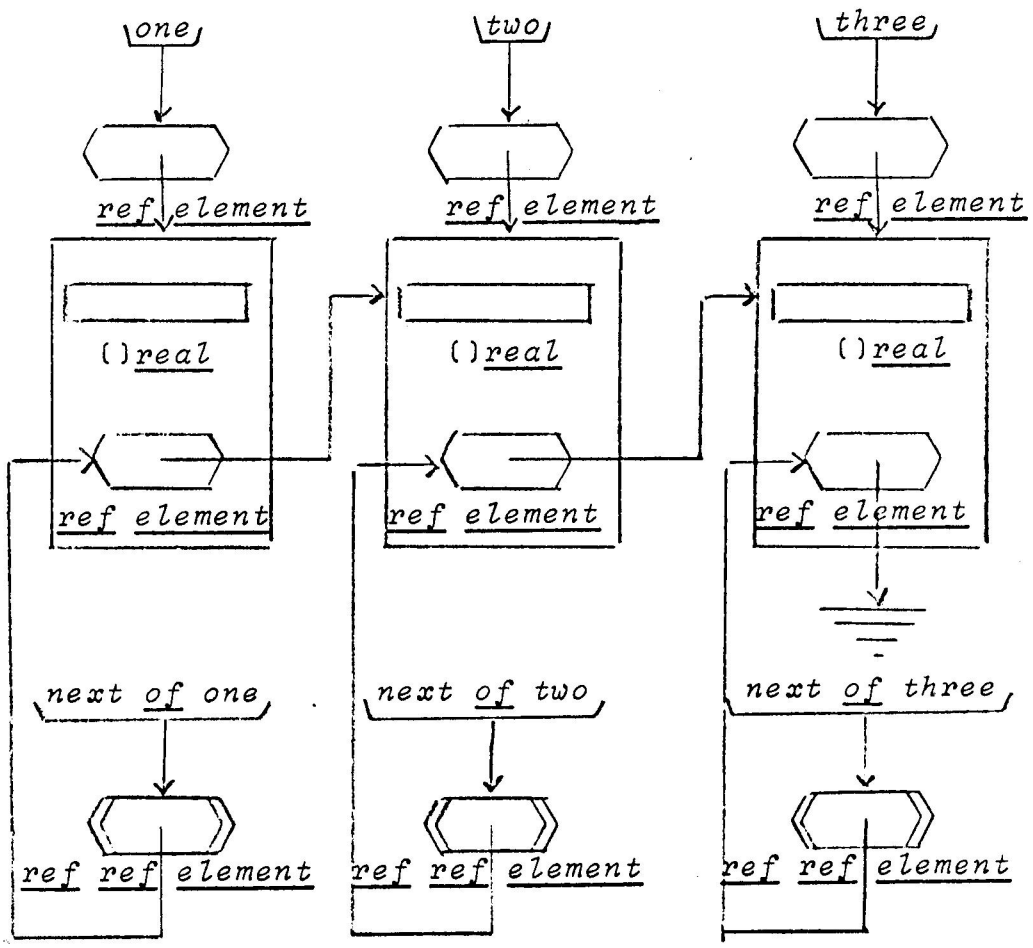
Deklariert man dann

element one , *two* , *three* ;

so können wir die drei *elements* zu einer linearen Liste verketten, indem wir schreiben:

```
next of one   := two   ;
next of two   := three ;
next of three := nil   ;
```

Im Bild können wir dies folgendermaszen zum Ausdruck bringen



Da in den meisten Fällen zu Beginn des Aufbaus einer Liste die Anzahl ihrer Elemente nicht bekannt ist, man auszerdem oftmals an eine bereits aufgebaute Liste weitere Elemente anfügen möchte, so wird man natürlich nicht für jedes einzelne Element einen eigenen identifier deklarieren können. Stattdessen kann man eine Prozedur deklarieren, die bei jedem ihrer Aufrufe ein neues Element erzeugt und an eine (vielleicht sogar leere) Liste anhängt. Wir wollen dabei eine Liste durch ihren Anfang und ihr Ende charakterisieren:

```
mode list  = struct ( ref element begin , end ) ;
```

Eine (noch leere) Liste erhalten wir dann durch:

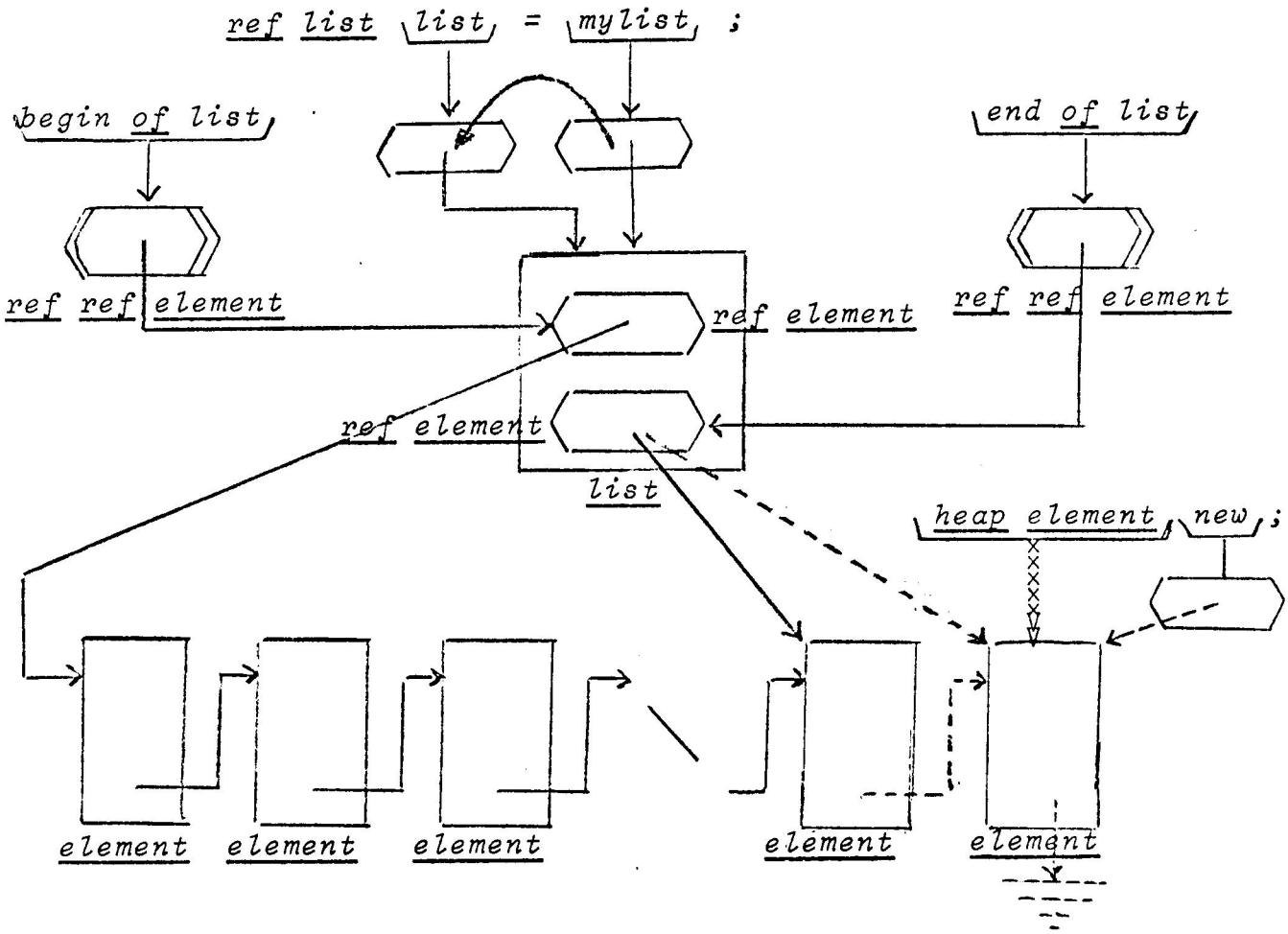
```
list mylist := ( nil , nil ) ;
```

Eine Prozedur, die ein neues *element* erzeugt und an eine aktuell übergebene Liste anhängt, wird gegeben durch:

```
proc generate = (ref list list) void:
                    ( heap element new ;
                        if begin of list is nil
                      then begin of list
                      else next of end of list
                        fi := new ;
                      end  of list := new ;
                      next of new  := nil
                    ) ;
```

Man beachte auch hier die Notwendigkeit des heap generator. Hätten wir den "locale" zur Aufnahme eines *element* auf dem "stack" erzeugt, so wäre er nach dem Aufruf von *generate* wieder von dem "stack" verschwunden. Wir haben deshalb diesen "locale" im "heap" erzeugt und seine Adresse in einen auszerhalb der Routine existierende "locale" transportiert.

Nach einem Aufruf *generate (mylist)* ergibt sich folgendes Bild:



Die unterbrochene Pfeile geben an, welche "references" durch den Aufruf *generate(mylist)* zustande gebracht werden.

Es lohnt sich, darüber nachzudenken, dasz wir *generate* auch auf folgende Weise deklarieren können:

```
proc generate = (ref list list) void:
                 ( heap element new ;
                   end of list :=
                        if begin of list is nil
                      then begin of list
                      else  next of  end of list
                        fi     :=   new ;
                  next of new  :=   nil
                 ) ;
```

Wie man sogar (wenn vielleicht auch weniger einsichtig) mit einer einzigen assignation auskommt, zeigt die folgende declaration

```
proc generate  =  (ref list list) void:
     next of ( end of list :=   if begin of list is nil
                                then begin of list
                                else  next of  end of list
                                  fi
                             := heap element )  :=  nil  ;
```

Wer die Programmiersprache LISP kennt, wird sich über solche, ineinander geschachtelte Konstruktionen nicht allzu sehr wundern.

Wie man ein aktuell übergebenes *element* (*this*) an eine bestimmte Stelle (*here*) innerhalb einer Liste einfügen kann, zeigt die folgende Prozedur:

```
proc insert  =  (ref element this , here) void:
     ( ref element tail  =  next of here ;
       next of here := this ;
       next of this := tail
     ) ;
```

Man beachte, dasz durch *insert* auch ein (schon existierendes) *element* an eine Liste angehängt werden kann. Es sei zum Beispiel deklariert:

```
list mylist ;  element thisone ;
```

Will man nun *thisone* am Ende von *mylist* anhängen, so schreibt man den Aufruf:

```
insert ( thisone , end of mylist )
```

Innerhalb der closed-clause von *insert* erhält dann der identifier *tail* den Wert *nil*.

4.4.3. Der cast

Wir haben schon mehrmals gesehen (vgl. 2.2.2 und 3.5.2), wie man mit Hilfe des cast explizit angeben kann, welchen "mode" der Wert einer enclosed-clause (vgl. 2.1.2) haben soll.

Ein cast hat die allgemeine Form:

$$\overbrace{\underline{amode}}^{\text{formal-declarer}} \quad \text{enclosed-clause}$$

Beispiele:

(1) *real(i)* und *real(j)*

in

y := real(i)×real(j) (vgl. 2.2.2)

(2) *ref real (yy)*

in

ref real (yy) := xx (vgl. 3.5.2)

(3) *compl if x≥0 then x else y fi*

in

z plusab compl if x≥0 then x else y fi

Der cast ist hier notwendig, da der Operator *plusab* zur Zuweisung an einen *ref compl* von seinem rechten Operanden einen *compl* fordert und "widening" in "firm context" nicht vorgenommen wird.

Durch einen cast wird also die enclosed-clause gezwungen, einen/durch den formal-declarer geforderten *amode*-Wert zu liefern.
Dies wird syntaktisch dadurch erreicht, dasz die enclosed-clause in einen "strong context" versetzt wird, damit alle "coercions" vorgenommen werden können. So gesehen, kann also ein cast als ein, alle "coercions" in sich vereinigender Anpassungsoperator aufgefaszt werden.

Eine wichtige (semantische) Rolle spielt der cast noch bei dem Aufruf einer Routine: er bestimmt stets den "mode" des Ergebnis-Wertes (vgl. 3.2.3 , 3.2.4 und 3.3.2) .

4.4.4. "Balancing"

"Balancing" bedeutet, dasz der Compiler ein unit das (auf sich betrachtet) nicht in einen "strong context" steht, auch ohne explizit angegebenen cast in einen "strong context" versetzt wenn er auf anderen Grund sicher sein kann, dasz dadurch keine Mehrdeutigkeiten (z.B. bei der Operator-Identifizierung, vgl. 5.1.4) auftreten können.
Wir betrachten dazu:

if $x \geq 0$ *then* 3.1415 *else* 1 *fi*

Steht diese conditional-clause in einen Kontext in dem kein "widening" vorgenommen wird (z.B. in einen "firm context"), so könnte der Compiler nicht entscheiden ob ihre "elaboration" einen Wert vom "mode" *real* bzw. *int* liefert. Von diese zwei Möglichkeiten kann jedoch die letzte sowieso ausgeschlossen werden: ALGOL68 kennt die "coercion" widentoreal aber nicht eine Anpassung 'narrowtoint'. Der Compiler versetzt dann hier die out-choice-clause *1* in einen "strong context" und übersetzt:

if deref $x \geq 0$ *then* 3.1415 *else* widentoreal 1 *fi*

Es sei deklariert:

int i ; *real* x ; *compl* u,z ;

Wir betrachten nun die formula

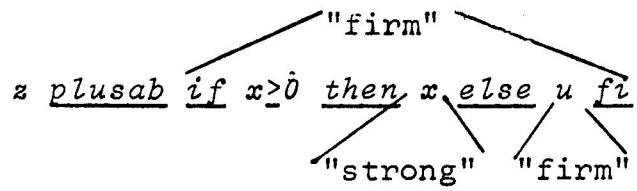
z *plusab* *if* $x \geq 0$ *then* x *else* u *fi*

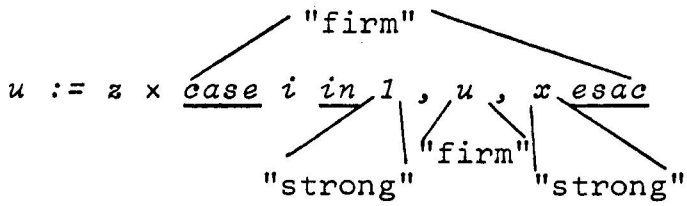
Wir wissen schon (vgl. 4.4.3), dasz der Operator *plusab* von seinem rechten Operand einen *compl* fordert. Dieser *compl* wird tatsächlich geliefert wenn zur "runtime" $x \geq 0$ den Wert *false* liefert. Deswegen kann der Compiler entscheiden, dasz der Programmierer einen *compl*-Wert als "yield" der "elaboration" nicht ablehnen wird; er versetzt somit die in-choice-clause in "strong context" und übersetzt:

z *plusab* *if* deref $x \geq 0$ *then* widentoreal deref x *else* deref u *fi*

Dieses Ausbalancieren des Kontextes beinhalt also, dasz, sobald der Compiler über den "mode" einer aus mehreren Alternatieven in nicht-"strong context" entscheiden kann, er alle andere Alternatieven in "strong context" versetzt, worin dann alle "coercions"

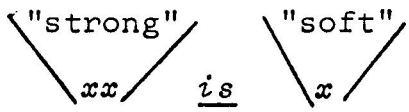
vorgenommen werden können:





Ein weiteres Beispiel für das "balancing" haben wir schon in 3.5.3 bei der identity-relation kennengelernt. Damit die *ref*-Stufe, auf der man zwei Adressen miteinander vergleicht, eindeutig ist, so steht immer eine der beiden Seiten einer identity-relation in einen "soft context". Auf diese Seite kann dann nur deproc vorgenommen werden, insbesondere also kein deref. Die jeweils andere Seite jedoch, kann dann von dem Compiler (da eben die *ref*-Stufe nun bekannt ist) ohne Gefahr der Mehrdeutigkeit in einen "strong context" versetzt werden, so dasz er alle "coercions" vornehmen kann.

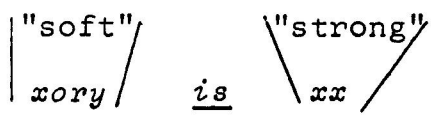
So wird die identity-relation *xx* *is* *x* (vgl. 3.5.3) auf folgender Weise ausbalanciert:



Der Compiler macht davon:

deref *xx* *is* *x*

Ein weiteres Beispiel (vgl. 3.3.3):



Also:

deproc *xory* *is* deref *xx*

4.5. Anwendungen und Beispiele

4.5.1 Ein 'switch'

Beim Programmieren kommt es häufig vor, dasz man einen Ansprungpunkt (gekennzeichnet durch einen label) im Programm mit Hilfe eines *int* 'berechnen' will; das heiszt, es hängt von dem *int*-Wert ab, zu welchem Punkt innerhalb des Programms gesprungen werden soll.
Wir haben schon gesehen, wie man dies mit Hilfe der case-clause (vgl. 2.6) sogar ohne Benutzung von labels erreichen kann. Will man jedoch (aus guten oder schlechten Gründen) durch einen jump zu einem 'berechneten' label springen, so erhebt sich die Frage, ob ALGOL68 (wie zum Beispiel ALGOL60) einen *switch* kennt. Die Antwort ist nein. Die bis jetzt behandelten Bausteine reichen jedoch aus, um (sogar auf mehrere Weisen) eine Art 'switch' zu ermöglichen.

Es seien im folgenden:

lab1: , *lab2:* , *lab3:* , *lab4:* , *lab5:* , - - - -
here: , *there:* , *again:* , *alarm:*

jeweils labels, die irgendwo innerhalb des Programms auftreten.

Zunächst sei bemerkt, dasz wir genau alles, was man in ALGOL60 zur Verfügung hat, erreichen können durch Prozedur-Deklarationen wie zum Beispiel:

```
proc jump = (int n) void:
    case n in lab1 , lab2 , lab3 , lab4 , lab5
          out alarm
    esac ;
```

Man sei daran erinnert, dasz die hier in der case-clause auftretenden units *lab1* , --- , *lab2* und *alarm* alle jumps sind (das goto-symbol braucht man nicht zu schreiben, vgl. 1.4.2). Durch den Aufruf *jump(i)* erreicht man dann, dasz entweder (wenn $1 \leq i \leq 5$) zu *lab1:* ,---, *lab5:* oder zu *alarm:* gesprungen wird.

Eine Konstruktion, die dem ALGOL60-*switch* noch näher kommt, ist die folgende:

(1:5) proc void jump = (lab1 , lab2 , lab3 , lab4 , lab5) ;

Man beachte, wie hier die source der identity-declaration einen row-display von *proc voids* liefert. Wenn wir nun ein Element auswählen durch *jump(i)*, so erhält dieser slice "a priori" einen *proc void*. Tritt er dann in einem Kontext auf, in dem das voiden vorgenommen werden kann, so wird nicht diese Routine selbst weggeworfen, sondern sie wird zunächst aufgerufen. Hat man also:

jump(i) ; unit

so macht der Compiler daraus:

deproc *jump(i) ;*

Es wird also der i-te jump ausgeführt.

An dieser Stelle sei folgendes angemerkt:

Hat man allgemein eine Prozedur ohne Parameter deklariert, also

proc procamode = amode: unit1

so macht der Compiler aus

procamode ; unit

stets (zu unserem Gefallen):

voiden deproc *procamode ;*

Weggeworfen wird demzufolge der Wert, den die Routine durch ihren Aufruf liefert, und nicht die Routine selbst. Dies gilt sogar auf jeder *proc*- und *ref*-Stufe. Erhält also zum Beispiel *p* einen *proc proc ref proc amode*, so macht der Compiler aus

p ; unit

stets

voiden deproc deref deproc deproc *p ;*

Will man unbedingt einen "mode" *switch* haben, so können wir in ALGOL68 sehr weit gehen. Man kann nämlich eine ()*proc void*-Variable deklarieren, wobei die Grenzen überdies flexibel sein können. So erhalten wir einen ganz besonderen "switch":

mode switch = *flex* (1:0)*proc void* ;

switch hop , *hip* , *hup* ;

Alle drei identifiers erhalten *ref switch*'s (sind also *switch*-Variable), an die irgendwelche Reihen von *proc void*'s, zum Beispiel von jumps, zugewiesen werden können:

hop := (*lab1* , *lab2* , *lab3* , *lab4* , *lab5*) ;
hip := (*lab1* , *lab2*) ;
hup := (*here* , *there* , *again*) ;

Sei nun i=3, so ist das Ergebnis von den Aufrufen:

hop(*i*) ⟶ *goto lab3*
hip(*i*) ⟶ eine "runtime"-Fehlermeldung (i> *upb hip*)
hup(*i*) ⟶ *goto again*

Man beachte wieder, wie hier

hup(*i*) ;

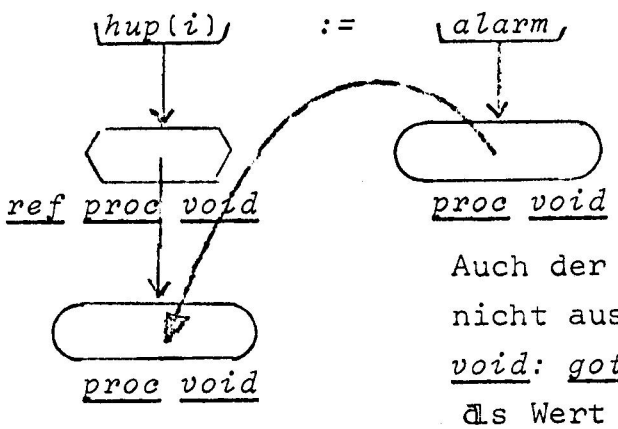
durch den Compiler übersetzt wird in

deproc deref *hup*(*i*) ;

Wenn aber zum Beispiel *hup*(*i*) als destination in einer assignation auftritt, zum Beispiel in

hup(*i*) := *alarm*

so wird hier *hup*(*i*) nicht angepaszt ("soft-context"). Dieser assignation entspricht folgendes Bild:



Auch der jump *goto alarm* wird nicht ausgeführt: es wird *void*: *goto alarm* (eine Routine) als Wert transportiert.

Nach den assignations

```
hup(3)    := alarm ;
hip       := hop   ;
hop(2:4)  := hup
```

ergibt sich durch die Aufrufe:

```
hup(3)  ---->  goto alarm
hip(3)  ---->  goto lab3
hop(3)  ---->  goto there
```

Die hier gemachten Ausführungen über verschiedene Arten von "switches" sollen aufzeigen, welche Möglichkeiten ALGOL68 bietet. Allerdings, die Möglichkeit des 'wilden' Springens innerhalb des Programms durch einen jump (man kann ja den label vor jedes unit innerhalb der unit-sequence einer serial-clause setzen) und viel mehr noch durch einen "switch", kann jedoch leicht zu einem völlig unstrukturierten Programm führen, welches dann gerade noch (höffentlich) der Programmierer selbst versteht. Deshalb vermeide man den jump, wo man nur kann. ALGOL68 bietet sehr viele Konstruktionen, die die Benutzung des jump überflüssig machen (conditional-clause, case-clause, loop-clause und alle ihre Kombinationen sowie das Konzept der Routine) und zu einer klareren Ausdrucksweise führen.

Wir hätten gern überhaupt nicht über jumps gesprochen, da jedoch in ALGOL68 der jump als einer der Grund-Bausteine vorkommt und man andererseits leider auch nicht gänzlich ohne ihn auskommt, so konnten wir dem Leser diesen schwarzen Baustein nicht einfach verschweigen.

4.5.2. Jensen-device

Eine für viele Anwendungen wichtige Technik ist der sogenannte 'Jensen-device': man deklariert eine Prozedur, bei deren Aufruf ein *ref amode*-Parameter eine Reihe von Werten durchläuft und andererseits dieser selbe Parameter als Argument in einer aktuellen Routine auftritt (Prinzip der gebundenen Variable).
Der 'Jensen-device' wird u.a. viel bei der Aufsummierung von Zahlen oder Funktionswerten verwendet. Die übliche mathematische Schreibweise hierfür ist:

$$\sum_{i=m}^{n} f(i)$$

In ALGOL68 können wir solch eine Prozedur folgendermaszen deklarieren:

```
proc sigma = (ref int i, int a,b, proc real fi) real:
    ( real value := 0 ;
      for k from a to b
       do i := k ;
          value plusab fi
       od ;
     value
    ) ;
```

Was hier passiert ist vielleicht nicht unmittelbar klar. Man beachte, dasz *fi* genau (b-a+1)-mal aufgerufen wird und im allgemeinen bei jedem Aufruf einen anderen Wert (hier ein *real*) als Ergebnis liefert. Weiter beachte man, dasz innerhalb der loop-clause die assignation *i:=k* ausgeführt wird.

Beim Aufruf von *sigma* wird also einem (aktuellen) *ref int*-Parameter jeweils einen Wert zugewiesen. Der control-identifier *k* ist hierbei aber lokal innerhalb der loop-clause.

Es sei deklariert

```
int p,q,m,n ;  read((m,n)) ;  real x,y ;
(1:n)real row ,  (1:n,1:m)real mat ;
proc  icos = (int i)real: cos(2xpixi/n) ;
proc ijcos = (int i,j)real: cos(2xpixi/j) ;
```

Wir betrachten nun

```
y := sigma ( p , 1,n , real: row(p) )
```

Das Wesentliche des 'Jensen-device' ist dann:

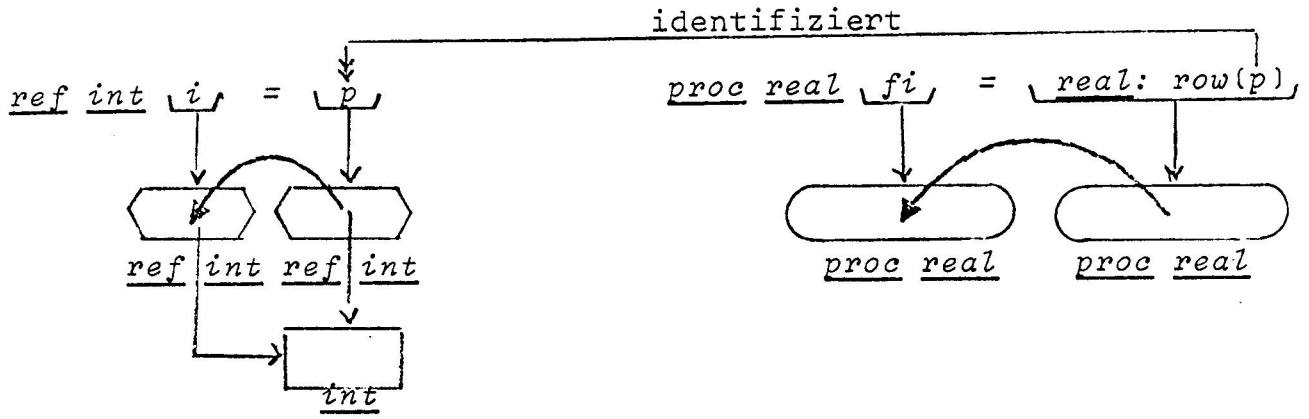
der aktuelle (*ref int*-)Parameter *p* tritt auch auf als Index in dem aktuellen routine-text *real: row(p)* ;

bei der "elaboration" von *sigma* wird der (*proc real*-) Parameter *fi*, also aktuell die Routine *real: row(p)* mehrmals aufgerufen.

Bei diesem Aufruf von *sigma* wird folgende closed-clause abgearbeitet:

```
(  ref int  i = p ,
       int  a = 1 , b = n ,
  proc real fi= real: row(p) ;
  real ( real value := 0 ;
           for k from a to b
            do i := k ;
               value plusab deproc fi
            od ;
          deref value
       )
)
```

Es lohnt sich die Mühe zu sehen, wie die beiden Relationen "to identify" und "to refer to" eng zusammenarbeiten, um das gewünschte Ziel zu erreichen:



Ein anderer Aufruf von *sigma* ist

y := sigma (p , -m,+m , real: icos(p))

und man sieht auch hier, wie der aktuelle (*ref int*) Parameter auftritt in dem routine-text *real: icos(p)* .

Eine für die Praxis interessante Anwendung findet der 'Jensen-device', wenn man ihn rekursiv über den (*proc real*-)Parameter verwendet:

y := sigma(p,1,n, real: sigma(q,1,n, real: mat(p,q)))

oder auch

y := sigma(p,m,n, real: sigma(q,m,n, real: ijcos(p,q))

Auch hier lohnt sich die Mühe, darüber nachzudenken, wie sich *sigma* bei seinem Aufruf über die formula

value plusab deproc *fi*

rekursiv aufruft (vgl. 3.4.5) .

In ALGOL68 kann man viele 'Jensen-artige' 'devices' konstruieren. So können wir die gebundene Variable *i* völlig implizit verwenden, wie man in folgender procedure-declaration sieht:

```
proc sum  =  (int a,b, proc(int)real fun) real:
      ( real value := 0 ;
         for k from a to b
          do value plusab fun(k) od ;
        value
      ) ;
```

Aufrufe von *sum* sind dann:

sum (1,n , (int i)real: row(i))

sum (1,n , icos)

Es ist vielleicht überraschend, dasz man in dem ersten Aufruf einen routine-text schreiben musz, jedoch in dem zweiten Aufruf der identifier genügt.

Auch *sum* kann man über den (*proc(int)real*-)Parameter rekursiv verwenden:

sum(1,n, (int i)real: sum(1,n, (int j)real: mat(i,j)))

sum(m,n, (int i)real: sum(m,n, (int j)real: ijcos(i,j)))

5. OPERATOREN UND FORMULAS

5.1 Das Konzept formula

5.1.1 Operatoren-Schreibweise und Prozedur-Schreibweise

5.1.2 Monadische und dyadische Operatoren (Prioritäten)

5.1.3 Die syntaktische Position von Operanden ("firm context")

5.1.4 Die Identifizierung eines Operators

5.1.5 Die "elaboration" von formulas

5.2 Operator-Definitionen aus der standard-prelude

5.2.1 Operatoren mit *bool*-Operanden

5.2.2 Operatoren auf arithmetischen Werten

5.2.3 Operatoren mit *compl* Operanden

5.2.4 Operatoren mit *string*-Operanden

5.2.5 Operatoren mit *bits*- und *bytes*-Operanden

5.2.6 Operatoren in Zusammenhang mit assignations

5.3 Beispiele von weiteren Operator-Definitionen

5. OPERATOREN UND FORMULAS

5.1. Das Konzept formula

5.1.1. Operatoren-Schreibweise und Prozedur-Schreibweise

Seit Jahrhunderten ist man in der Mathematik daran gewöhnt, Operatoren in der Weise zu benutzen, dasz sie zwischen ihren beiden Operanden stehen (Infix-Form). So zum Beispiel:

$$a + b \times c \uparrow 2 - d/e$$

Bei solch einer Operatoren-Schreibweise wird auszerdem jedem Operator auf eindeutige Weise eine bestimmte Priorität zugeordnet. Der praktische Vorteil liegt darin, dasz man sich eine explizite Klammerung bei der Abarbeitung einer aus mehreren Operatoren bestehenden Formel ersparen kann. Unser Beispiel hat dann die folgende implizite (durch Prioritäten gegebene) Klammerung

$$(a+(b\times(c\uparrow 2)))-(d/e)$$

Durch eigene (explizit angegebene) Klammerung kann man jedoch die vorgegebenen Prioritäten ändern:

$$(a+b)\times(c\uparrow 2-d)/e$$

Man vergleiche auch

$a + b\times c$	mit	$(a+b)\times c$
$a - b + c$	mit	$a - (b + c)$

Beim letzten Beispiel wird durch die explizite Klammerung

$$a - (b + c)$$

die übliche und in vielen Programmiersprachen verwendete Links-Rechts-Verarbeitung von Operatoren gleicher Priorität:

$$(a - b) + c$$

durchbrochen.

Man könnte nun andererseits jeden Operator durch eine entsprechende Prozedur ersetzen, so dasz dann eine Formel statt durch die Operatoren-Schreibweise durch ineinander geschachtelte Prozedur-Aufrufe dargestellt werden kann. Hierbei musz man dann allerdings die Prioritäten der einzelnen Operationen explizit zum Ausdruck bringen. Für die zuerst angegebene Formel könnte man z.B. die folgende Prozedur-Schreibweise benutzen:

$$sub(add(a,mul(b,pow(c,2))),div(d,e))$$

Für den Menschen ist diese Schreibweise jedoch etwas schwer zu lesen. Ein Computer dagegen würde darüber sehr glücklich sein. Deshalb wandeln viele Compiler die obige, in der Operatoren-Schreibweise gegebene Formel in die angegebene Prozedur-Schreibweise um und können dann sogar auf die Klammern verzichten:

$$\underline{sub}\ \underline{add}\ a\ \underline{mul}\ b\ \underline{pow}\ c\ 2\ \underline{div}\ d\ e \qquad \text{(Prefix-Form)}$$

Noch glücklicher wäre der Computer wenn der Compiler die Formel gleich in die Postfix-Form umwandelt:

$$a\ b\ c\ 2\ \underline{pow}\ \underline{mul}\ \underline{add}\ d\ e\ \underline{div}\ \underline{sub} \qquad \text{(Postfix-Form)}$$

Wie gesagt, der Vorteil der Operatoren-Schreibweise in der Infix-Form gegenüber die Prozedur-, bzw. Prefix- oder Postfix-Schreibweise liegt in ihrer gröszeren 'Menschenfreundlichkeit', die ihre Ursache jedoch allein in der Gewohnheit findet. Zu dieser Gewohnheit gehört auch noch die merkwürdige Tatsache, dasz man dasselbe Operator-Zeichen für zwei verschiedene Operatoren, nämlich einmal für einen dyadischen und einmal für einen monadischen Operator verwenden kann. Ein jeder versteht zum Beispiel:

$$-a + b \qquad \text{und} \qquad a \times -b$$

Hierbei wird das Zeichen "-" in beiden Fällen als monadischer Operator verwendet. Solch ein monadischer Operator hat dabei nur einen (rechten) Operanden.
Die Operatoren-Schreibweise hat man sogar auch über die Mathematik hinaus in andere Bereiche getragen. So kann zum Beispiel

$$u \times v$$

bedeuten:

a) Multiplikation zweier ganzer Zahlen ,
b) Multiplikation zweier reeller Zahlen ,
c) Äuszeres Produkt zweier Vektoren ,
d) Multiplikation zweier Matrizen ,
e) Kartesisches Produkt zweier Mengen ,
f) Logische Konjunktion zweier Aussagen ,
g) Astrologische Konjunktion zweier Planeten ,
h) Heirat zwischen Mann und Frau ,
i) Das Schlagen eines Steines durch einen anderen (Schachspiel) .

Ganz allgemein bedeutet $u \times v$ also: irgendeine Verknüpfung zweier Elemente in irgendeiner strukturierten Menge (z.B. Gruppe, Ring, Körper, Menschheit, Brettspiel usw.).

Mathematiker und Techniker sind an diese Operatoren-Schreibweise so gewöhnt, dasz sie von einer höheren Programmiersprache erwarten, diese in ihr benutzen zu können.
Um also diesen Gewohnheiten entgegen zu kommen, so kann man auch in ALGOL68 diese Operatoren-Schreibweise benutzen.
Dies geschieht durch Verwendung der formulas. Dabei kann eine formula entweder eine dyadische oder monadische sein:

formula { dyadic - formula ⟶ operand operator operand
monadic- formula ⟶ operator operand

Dabei kann jeweils ein operand selbst wieder eine dyadic- oder monadic-formula sein, wir erhalten somit die üblichen mathematischen Formeln:

a+b , *-a+b* , *ax-b* , *a+bxc↑2-d/e*

sind jeweils dyadic-formulas.
Dagegen sind:

-a , *sign entier -(a+b)* , *- abs round re z*

jeweils monadic-formulas.

Eine formula enthält also mindestens einen operand und mindestens einen operator.
Die "elaboration" einer formula geschieht durch die "elaboration" der in ihr enthaltenen formulas. Dabei ist die Reihenfolge der "elaboration" durch die vorkommenden operators bestimmt:

monadic-formulas werden zuerst abgearbeitet,
sodann die dyadic-formulas von der höchsten zur niedrigsten Priorität.

Allerdings, durch explizite Klammerung kann man diese Reihenfolge jederzeit ändern.

5.1.2. Monadische und dyadische Operatoren (Prioritäten)

Wie wir in 2.4.1 schon gesehen haben, hat man in ALGOL68 die Möglichkeit, neben identifiers und mode-indications auch Operatoren und deren Prioritäten selbst zu definieren.
Dabei gilt für einen Operator stets zweierlei:

1) er hat eine bestimmte Priorität (durch eine priority-declaration)
2) er erhält enie Routine als Wert (durch eine operation-declaration

Es gibt zwei verschiedene Formen von operation-declarations, je nachdem ob sie durch ein monadischer oder dyadischer Operator definiert wird:

operation-declaration:

- mono-operation-declaration: *op*(*amode1*)*amode* (formal-plan) defining-operator = mono-unit / source
- duo -operation-declaration: *op*(*amode1*,*amode2*)*amode* (formal-plan) defining-operator = duo -unit / source

Es musz in beiden Fällen die source eine Routine liefern, die dann der defining-operator erhält. Dabei musz im Falle einer mono-operation-declaration diese Routine ein *proc*(*amode1*)*amode* und im Falle einer duo-operation-declaration ein *proc*(*amode1*,*amode2*)*amode* sein. Im ersten Fall wird also ein dyadischer Operator und im zweiten Fall ein monadischer Operator definiert. Welche Routine die source zur "runtime" liefern musz, wird schon dem Compiler durch den formal-plan mitgeteilt. Der formal-plan gibt darüberhinaus an, von welchem "mode" der Wert ist, den die "routine" bei ihrem Aufruf liefern musz.

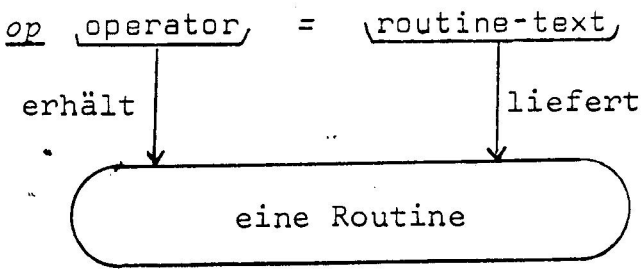
Als Beispiel betrachten wir die beiden operation-declarations durch die der monadische Operator *not* und der dyadische Operator *and* (jeweils für *bool*-Operanden) standardmäszig in ALGOL68 definiert worden sind (man vergleiche hier 1.3.4):

op (*bool*)*bool* *not* = (*bool* *a*) *bool*: *if* *a* *then* *false* *else* *true* *f*
op(*bool*,*bool*)*bool* *and* = (*bool* *a*,*b*)*bool*: *if* *a* *then* *b* *else* *false* *fi* ;

Hier wird in beiden Fällen die source durch ein routine-text gegeben. Da ein routine-text bereits selbst genau den "mode" seiner Routine beschreibt, so ist der formal-plan auf der linken Seite überflüssig. Das symbol *op* allein reicht hier aus, wir können also kürzer schreiben:

op not = (*bool* a)*bool*: *if* a *then false else true fi* ;
op and = (*bool* a,b)*bool*: *if* a *then* b *else false fi* ;

Man beachte hierbei die auffallende Ähnlichkeit mit einer procedure-declaration (vgl. 3.2.2). Auch hier können wir die "elaboration" solch einer operation-declaration durch ein Bild darstellen:



durch den routine-text selbst bestimmt:

proc(*amode1*)*amode*
oder *proc*(*amode1*,*amode2*)*amode*

Jeder Operator hat eine bestimmte Priorität. Dabei haben alle monadischen Operatoren die gleiche Priorität 10 und für die dyadischen Operatoren kann man die Prioritäten 1 bis 9 in einer priority-declaration bestimmen. Zum Beispiel:

prio and = 3

Diese priority-declaration finden wir in der standard-prelude. Darüberhinaus sind in der standard-prelude die Prioritäten aller gebräuchlichen Operatoren festgelegt. Einige davon sind im folgenden angegeben:

prio minusab = 1 , *plusab* = 1 , *timesab* = 1 , *divab* = 1 ,
or = 2 , *and* = 3 , = = 4 , ≠ = 4 , < = 5 , ≤ = 5 ,
> = 5 , ≥ = 5 , − = 6 , + = 6 , × = 7 , *over* = 7 ,
mod = 7 , / = 7 , ↑ = 8 , *lwb* = 8 , *upb* = 8 , *i* = 9 ;

Man beachte hierbei, wie man für mehrere Operatoren ihre Prioritäten in einer einzigen priority-declaration festlegen kann.

Die Abarbeitung der formula

$$a + b \times c \uparrow 2 - d / e$$

geschieht dann so, als ob wir explizit geklammert hätten:

$$(a+(b\times(c\uparrow 2)))-(d/e)$$

Also, die Operatoren höchster Priorität werden absteigend zu den Operatoren niedrigsten Priorität abgearbeitet.

Der Programmierer hat jedoch die Möglichkeit, z.B. in einer serial-clause, die Prioritäten von Operatoren zu ändern. So kann er zum Beispiel den Wunsch haben, alle oben angegebenen Operatoren mit der gleichen Priorität zu versehen. Er kann dann schreiben:

$$\underline{prio}\ + = 6\ ,\ - = 6\ ,\ \times = 6\ ,\ \uparrow = 6\ ;$$

Sodann wird unsere formula so abgearbeitet, alsob wir folgendermaszen geklammert hätten:

$$((((a+b)\times c)\uparrow 2)-d)/e$$

Also, Operatoren gleicher Priorität werden von links nach rechts abgearbeitet.

Da die monadischen Operatoren die höchste Priorität haben, so werden diese innerhalb von formulas stets zuerst abgearbeitet, zum Beispiel:

$$-x\uparrow 2 \quad , \quad -n + \underline{abs}\ x$$

werden abgearbeitet als hätten wir geklammert:

$$(-x)\uparrow 2 \quad , \quad (-n)+(\underline{abs}\ x)$$

WARNUNG: ins besondere bei $-x\uparrow 2$ unterscheidet sich ALGOL68 wesentlich von ALGOL60

Treten mehrere monadische Operatoren hintereinander auf, wie zum Beispiel in

$$\underline{abs}\ \underline{round}\ \underline{re}\ z$$

so werden diese natürlich so abgearbeitet, als hätten wir geklammert:

$$(\underline{abs}(\underline{round}(\underline{re}\ z)))$$

Wie wir den Tabellen für die Standard-Operatoren in 1.3.4 entnehmen können, kann durchaus der gleiche Operator für verschiedene Operanden definiert sein. Seine ihm einmal zugeteilte Priorität bleibt davon unberührt.
So ist zum Beispiel der dyadische Operator × für alle möglichen Kombinationen von *ints*, *reals* und *compls* definiert (vgl. 1.3.4).

Will man sich die Prozedur bei der Trauung zwischen *man* und *woman* ersparen, so kann sie auch weniger aufwendig mit Hilfe einer einfache formula verheiraten. Dazu können wir dann den Operator × in folgender duo-operation-declaration ein weiteres Mal definieren:

```
op × = (ref man he , ref woman she) void:
        (   wife of he := she ,
         husband of she:= he   )
```

Will man dann z.B. *henry8* mit *anna boleyn* verheiraten, so braucht man nur die formula

```
henry8 × anna boleyn
```

zu schreiben und alles ist perfekt (vgl. auch 3.3.2); für die "elaboration" von formulas vergleiche 5.1.5.

5.1.3. Die syntaktische Position von Operanden ("firm context")

Um die Möglichkeit zu haben, den gleichen Operator für Operanden von verschiedenen "mode" zu verwenden (der Operator erhält somit, abhängig von dem "mode" seiner Operanden), so kann man nicht alle Anpassungsoperationen auf Operanden zulassen, da sonst bei der Identifizierung des Operators Mehrdeutigkeiten auftreten könnten (vgl. 5.1.4).
Wie wir aus dem Überblick von 4.4.2 ersehen können, stehen die Operanden in einer formula stets im "firm context". Dies bedeutet, dasz der Compiler zwar das "deproceduring", "dereferencing" und "uniting" anwenden wird, nicht aber das "widening", "rowing" und "voiding".
Betrachten wir zum Beispiel die formula

$$i + j$$

Würde man hier auf die Operanden i und j das "widening" zulassen, so wüszte man nicht, ob zum Beispiel die beiden *int*-Werte von i und j oder die beiden *real*-Werte, die i und j nach dem "widening" liefern, addiert werden.
Der Programmierer könnte dann also nicht davon ausgehen, dasz die von ihm gewünschten Operationen auch tatsächlich ausgeführt werden, obwohl er doch alles in seiner Macht stehende dafür getan hat, nämlich die seinen Wünschen entsprechende operation-declaration zu schreiben.

5.1.4. Die Identifizierung eines Operators

Die Möglichkeit, den gleichen Operator für Operanden von verschiedenem "mode" zu definieren, bedeutet also, dasz dieser Operator verschiedene Routinen erhalten kann, abhängig von dem "mode" seiner Operanden. Insbesondere bedeutet es, dasz dieser Operator sowohl monadisch als dyadisch definiert sein kann.
Tritt dann solch ein mehrmals deklarierter Operator als "applied-operator" in einer formula auf, so erhebt sich die Frage, welcher der vorhandenen "defining operators" identifiziert wird, d.h. welche "property" der "applied-operator" erhält (vgl. 2.4.1 und 2.4.2).
Betrachten wir dazu folgendes Beispiel:

Man kann durchaus verschiedene Auffassungen über Gleichheit von arithmetischen Werten haben. Darüberhinaus braucht diese Gleichheit für die einzelnen Typen nicht dieselbe zu sein.
Im folgenden verwenden wir das Zeichen "?" sowohl als monadischen als auch als dyadischen Operator, einmal, um bei *real*s ihren gebrochenen Teil und bei *compl*s ihre Richtung anzugeben, zum andern, um solch eine Gleichheit auszudrücken.

```
op ? = (real a)  real: a - entier a ;
op ? = (real a,b)bool: ?a = ?b ;
op ? = (compl a)  real: arg a ;                    (vgl. 5.2.3)
op ? = (compl a,b)bool: ?a = ?b ;
op ? = (real a , compl b)bool: im b = 0 ;
op ? = (compl a , real b)bool: b ? a ;
```

Betrachten wir dann im Bereich der declarations *real* x,y und *compl* u,v die folgenden formulas:

```
x ? y  ,  x ? u  ,  u ? x  ,  u ? v  ,

x ? ?u ,  u ? ?x ,  ?u ? ?x ,  ?u ? ?v
```

Man beachte hierbei die implizite Klammerung innerhalb der formulas.

Hierbei haben wir durch Pfeile angedeutet, welcher "applied operator" welchen "defining operator" identifiziert. Zunächst wird in der kleinsten serial-clause, die den "applied operator" enthält nach einem oder mehreren "defining operators" gesucht, die durch das gleiche Operator-Zeichen dargestellt werden wie der "applied operator" (hier "?"). Führt die Suche hier zu keinem Erfolg, so wird in der,die jeweils bisher untersuchte serial-clause umfassenden serial-clause weitergesucht.

Hat man auf diese Weise einen oder mehrere "defining operators" gefunden, so werden die "modes" der Operanden (bzw. des Operanden) des "applied operator" mit den "modes" der entsprechenden formalen Parameter der gefundenen "defining operators" verglichen. Es wird dann derjenige "defining operator" identifiziert, bei dem die "modes"seiner formalen Parameter entweder mit den "modes" der Operanden übereinstimmen oder aber die "modes" der Operanden in die "modes" der entsprechenden formalen Parameter mit Hilfe der drei "coercions" deref, deproc und unite (nur diese werden im "firm context" vorgenommen) übergeführt werden können.
Stimmen die "modes" der Operanden mit den "modes" der formalen Parameter nicht überein und ist dies auch nicht durch deref, deproc oder unite zu erreichen, so wird innerhalb der betrachteten serial-clause keiner der "defining operators" identifiziert und die Suche wird dann in der umfassenden serial-clause fortgesetzt. Hierbei gibt es jedoch eine Ausnahme, die wir am Schlusz dieses Paragraphen behandeln.

Wie bei mode-indications und identifiers (vgl. 2.4.2) führt auch hier die Suche spätestens in der äuszersten serial-clause zum Erfolg, es sei denn, der Operator ist standardmäszig (in der standard-prelude) definiert.

Was bisher gesagt wurde, ist jedoch noch nicht die ganze Wahrheit; denn die bisher geschilderte Identifizierung kann immer noch mehrdeutig sein. Dies illustriert folgendes Beispiel:

(a) *op ? = (real a,b)bool: ?a = ?b ;*

(b) *op ? = (ref real a,b)bool: a is b ;*

Nach dem bisher Gesagten identifiziert der "applied opderator" ? in der formula *x?y* sowohl den "defining operator"(a) als auch den "defining operator" (b).

Um solche Mehrdeutigkeiten bei der Identifizierung von vornherein zu verhindern dürfen in dem Gültigkeitsbereich eines Operator (z.B. in ein und derselben serial-clause) nicht solche operation-declarations auftreten, bei denen die "modes" der sich jeweilig entsprechenden formalen Parameter durch Voraussetzen der 'mode-maker' *ref* und *proc* an *amode* ergeben.
Wir wissen schon (vgl. 4.3.1), dasz solche "modes" zu eng miteinander verbunden sind; denn sie können durch mehrmaliges deref und deproc wieder ineinander übergeführt werden.

Da auf Operanden ("firm context") auch die "coercion" unite vorgenommen werden kann, so darf darüberhinaus einer der "modes" der sich entsprechenden formalen Parameter zweier operation-declarations nicht erhalten werden durch Anwendung von unite auf *amode*, welcher so eng (wie eben besprochen) mit dem anderen "mode" der sich entsprechenden formalen Parameter verbunden ist.
Betrachten wir dazu das folgende Beispiel:

```
(a)   op ? = (ref real a,b)bool: a is b ;          (vgl. 3.5.3)

(b)   op ? = (union(real,compl) a,b)bool:
                 case a
                   in (real x): case b
                                  in (real  y): x ? y  ,
                                     (compl v): x ? v
                                esac  ,
                      (compl u): case b
                                  in (real  y): u ? y  ,
                                     (compl v): u ? v
                                esac
                 esac  ;
```

Auch hier ist also die Identifizierung des "applied operator" ? in der formula *x?y* nicht eindeutig, da auszer (a) auch (b) identifiziert werden kann (nämlich durch unite deref *x* ? unite deref *y*).

So, mit einiger Mühe haben wir (mitschwimmend im Sog des Reports) die bei der Identifizierung von Operatoren möglichen Mehrdeutigkeiten ausgeschlossen.

Allerdings bleibt noch einiges unschön, wie uns das folgende Beispiel zeigt:

```
begin   op ? = (real a)real: a - entier a ;

        begin
            op ? = (ref real a)real: entier a + 1 - a ;

        real x := 3.1415 ;

            print ( ( ?x , ?3.1415 ) )

        end

end
```

Nachdem, was wir bisher über die Identifizierung gesagt haben, identifiziert der "applied operator" *?* in der formula *?x* einen anderen "defining operator" als der "applied operator" *?* in der formula *?3.1415*. Der Programmierer würde sich wahrscheinlich sehr wundern, wenn hier gedruckt würde:

0.8485 0.1415

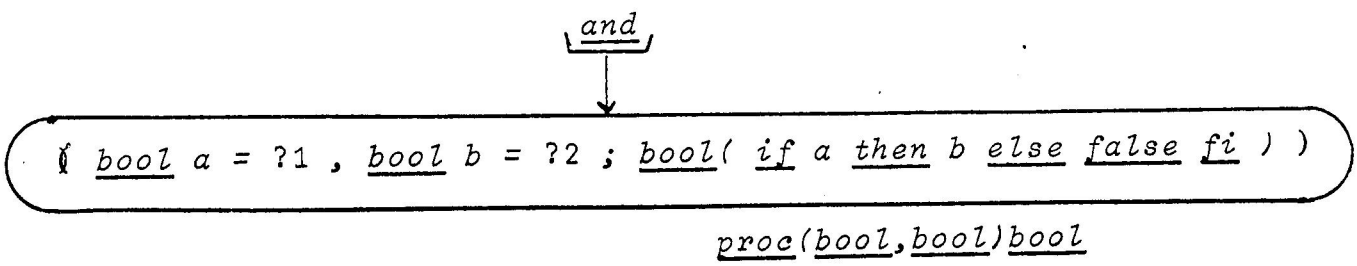
Da dies also mehr eine Falle für den Programmierer als ein sinnvolles Konzept ist, so wird dies von der Syntax her vermieden, indem gefordert wird: wird bei der Identifizierung eines "applied operator" eine operation-declaration gefunden, bei der die "modes" ihrer formalen Parameter mit den "modes" der Operanden der formula so eng wie vorher erwähnt miteinander verbunden sind, die "modes" der Operanden aber andererseits nicht durch deref, deproc oder/und unite in die "modes" der formalen Parameter übergeführt werden können (die oben auftretende Situation), so wird der Identifizierungsprozesz abgebrochen und von dem Compiler eine entsprechende Fehlermeldung ausgegeben. Das oben angegebene Programmm ist somit syntaktisch inkorrekt.

5.1.5. Die "elaboration" einer formula

Betrachten wir zunächst eine einfache formula, d.h. eine solche, die keine weiteren formulas enthält, zum Beispiel:

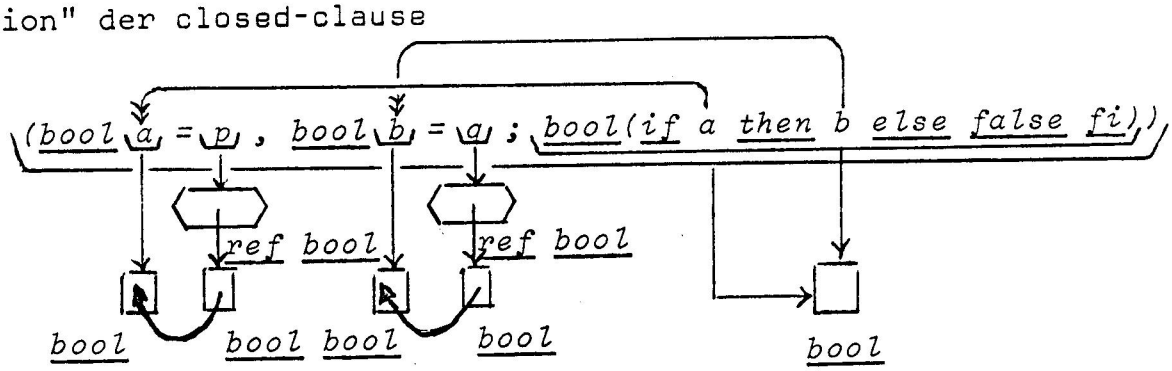
p <u>*and*</u> *q*

Ihre "elaboration" beginnt dann mit der "collateral elaboration" der beiden Operanden.
Der Operator <u>*and*</u> erhält die folgende Routine:



Es werden sodann an die Routine, die der Operator erhalten hat, die Werte der Operanden übergeben, und daran anschlieszend wird diese Routine aufgerufen. Der Wert, den die Routine bei ihrem Aufruf liefert, ist dann der Wert der formula.

Die "elaboration" von *p* <u>*and*</u> *q* geschieht also durch die "elaboration" der closed-clause



Die "elaboration" solch einer einfache formula entspricht also völlig der "elaboration" des call einer Prozedur (mit genau einem oder genau zwei formalen Parameter, vgl. 3.2.4 und 3.3.2), wobei man die Operanden als aktuelle Parameter ansehen kann. Man beachte, wie also auch hier wieder die Parameterübergabe durch entsprechende identity-declarations vorgenommen wird, in unserem Beispiel durch

<u>*bool*</u> *a* = *p* und <u>*bool*</u> *b* = *q*

Eine formula kann nun aus weiteren formulas bestehen (wie auch die aktuellen Parameter einer Prozedur selbst wieder calls von anderen Prozeduren sein können).
Zum Beispiel:

not (p and not q)

Die Prioritäten der in der formula auftretenden Operatoren geben an, in welcher Reihenfolge die Operatoren abgearbeitet werden, in unserem Beispiel also:

not (p and (not q))

Man beachte hierbei, dasz der Compiler diese implizite Klammerung unabhängig von den "modes" der Operanden vornehmen kann und musz ("mode-independent parsing"). Erst dann kann die Identifizierung der Operatoren stattfinden; denn die "modes" der Operanden spielen hierbei eine entscheidende Rolle (vgl. 5.1.4).
Durch diese (implizite) Klammerung wird also die gesamte formula in einfache formulas zerlegt, die selbst nur aus einem einzigen Operator bestehen.
Die "elaboration" von *not (p and not q)* geschieht also durch die "elaboration" der folgenden closed-clause:

```
( bool a = ( bool a = p ,
             bool b = ( bool a = q ;
                        bool ( if a then false else true fi )
                      ) ;
             bool ( if a then b else false fi )
           ) ;
  bool ( if a then false else true fi )
)
```

Im allgemeinen wird eine formula einen *nonvoid* liefern (nur dann kann sie in einer anderen formula als Operand auftreten). Eine formula darf jedoch einen *void* liefern, wie wir schon bei der Trauung *henry8 × anna boleyn* gesehen haben (vgl. 3.3.2 und 5.1.2). Diese Heirat kann also nicht mehr als Operand in einer anderen Heirat auftreten; sie liefert *void*, obwohl sicher einiges bei dieser Heirat passierte.